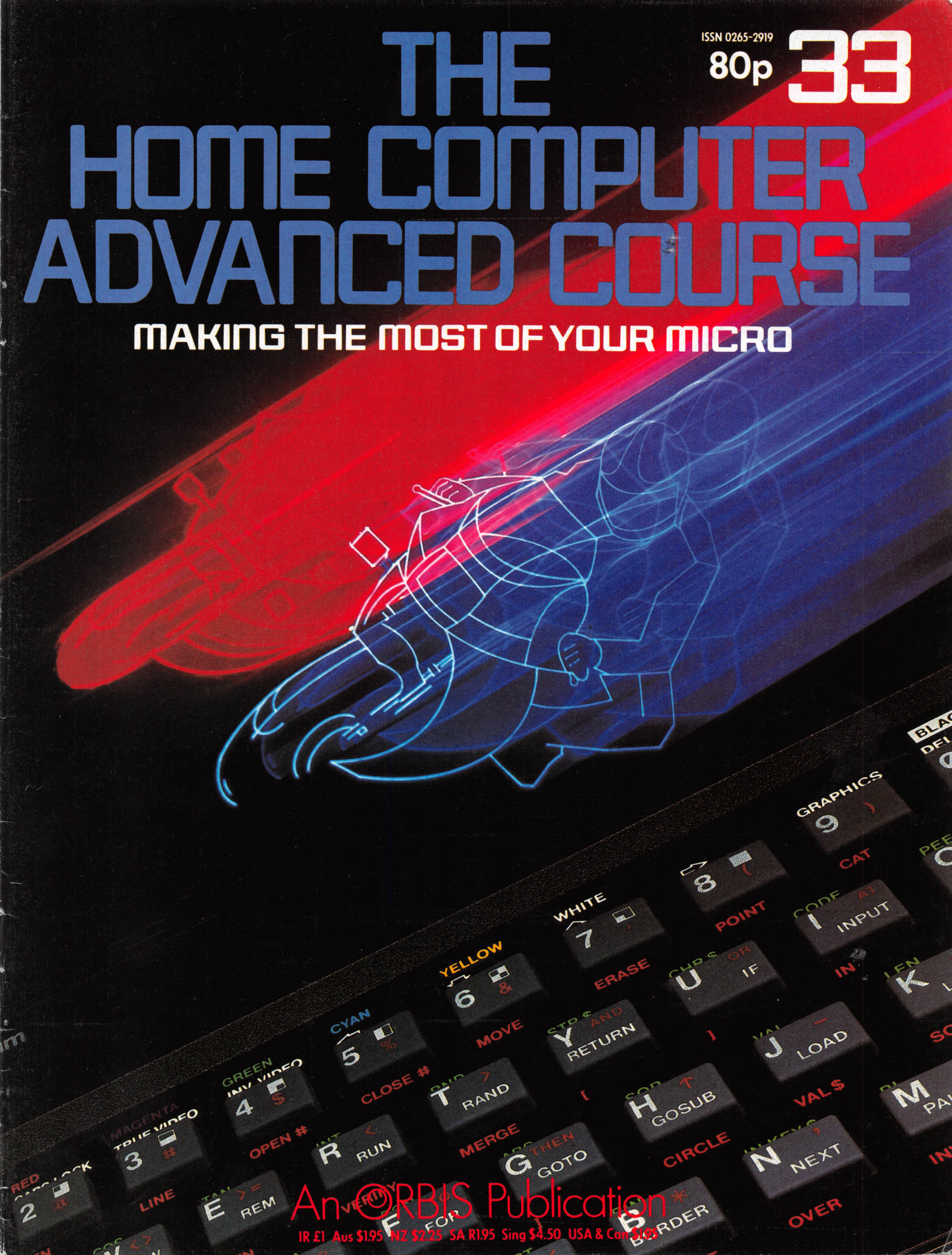
ISSN 0265-2919

80p 33

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **In our LOGO course, we move from the general to the particular with a look at LOGO sprites on the Commodore 64**
- **Our robotics series continues with a discussion of robot arms**
- **Having discussed the conventional, multi-program approach to integrated software, we consider the alternatives. In particular, we look at the unique software developed for Apple's Macintosh and Lisa**

QUIZ

1) For what purpose are round brackets used in LOGO?
2) What is postfix notation?
3) Which company manufactures the M10 microcomputer?
4) What is a 'keyboard macro'?

Answers To Last Week's Quiz

1) A floppy disk is said to be 'hard-sectored' if it has a number of index holes punched around its inner rim, marking the sector boundaries on each track.
2) MIRROR is a graphics option on Audiogenic's Koala-pad for the Commodore 64.
3) In an integrated software package, 'commonality' is the shared user image of the constituent packages — in other words, they all look and 'feel' the same.
4) Robots that replace human workers, especially in heavy engineering industries, are known as 'metal collar workers'.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Chris Naylor, Max Phillips, Steve Malone; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editors** Maurice Geller, Chris Cooper; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE, Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE, Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

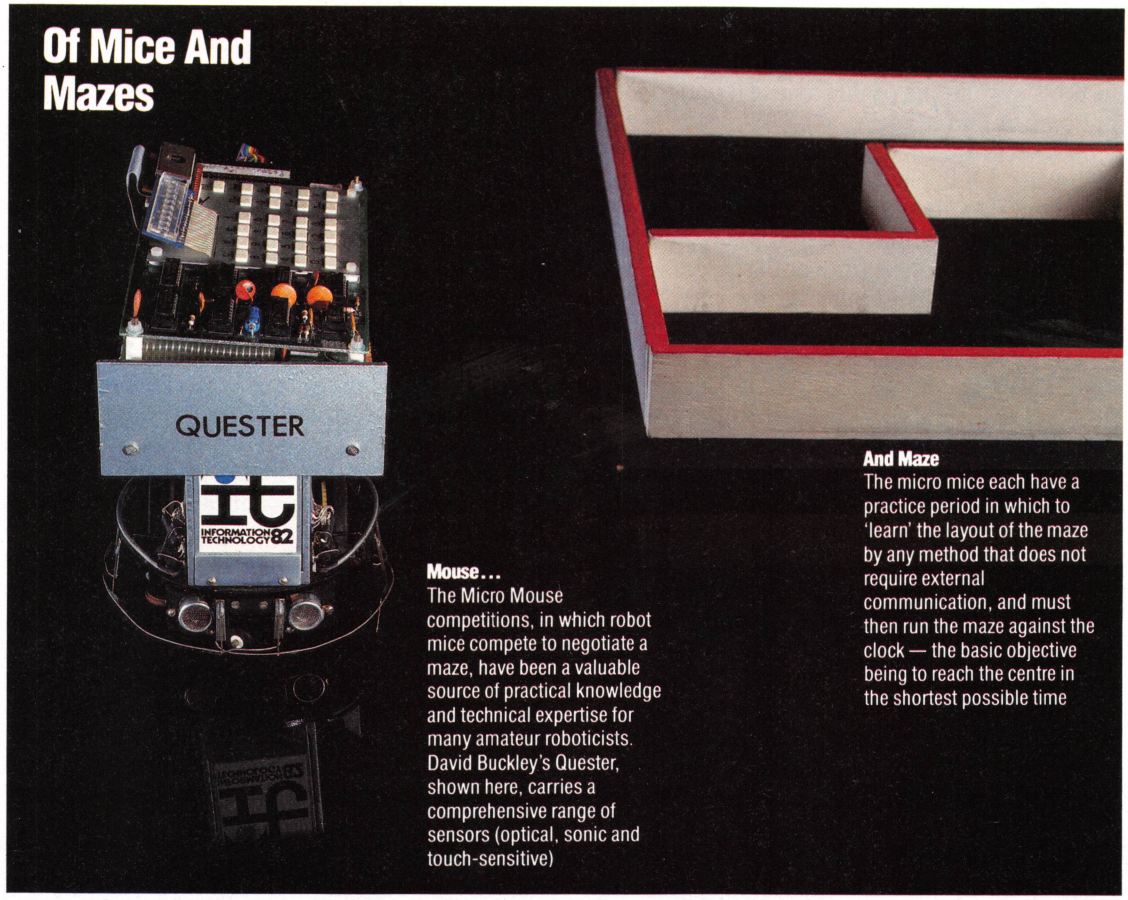
How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

COVER PHOTOGRAPHY BY PAUL CHAVE

THE STRAIGHT AND NARROW

Of Mice And Mazes



Mouse...
The Micro Mouse competitions, in which robot mice compete to negotiate a maze, have been a valuable source of practical knowledge and technical expertise for many amateur roboticists. David Buckley's Quester, shown here, carries a comprehensive range of sensors (optical, sonic and touch-sensitive)

And Maze
The micro mice each have a practice period in which to 'learn' the layout of the maze by any method that does not require external communication, and must then run the maze against the clock — the basic objective being to reach the centre in the shortest possible time

MARCUS WILLY

We have already looked at the three principal methods of robot movement (see page 621) and shown why electric motors are the most commonly used. Once in motion, however, a robot needs to be made to move where we want it to. Here we investigate ways of controlling a robot's movement.

The simplest method of moving a robot around involves using a mechanical device that 'reads' a specially-shaped card inserted into the robot. The outline of the card is followed by a small cam, which in turn operates a series of levers to control the robot's direction. In the past it was possible to buy model cars and small toy robots that operated in this way. A program was created using a pair of scissors to cut a card in the required shape. The robot would move according to the jagged edge.

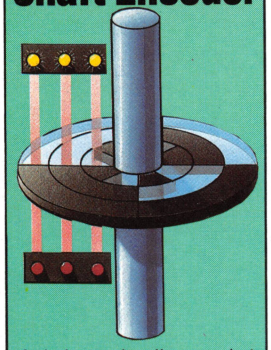
Other robots used devices that allowed them to follow a set route by means of internal electromechanical relays. These mechanical methods of movement control, however, were limited in application for the simple reasons that mechanical parts tend to be expensive and relatively inaccurate. But they do provide a precedent for contemporary methods.

One of the better methods now used involves a robot following a track specially laid for it on the ground. This is similar to the method used by model racing cars,which have a guide pin inserted into a continuous slot in a model racing track. The two most common forms of track-following robots, however, are those that follow a line drawn on the ground and those that are guided by a wire.

Robots following a line do so by using a light sensor — typically a photoelectric cell or an infrared sensor — to determine whether the robot is standing over a 'light' area or a 'dark' area. If the background colour of the ground is dark and the line is light, the output from the sensor will always be at its highest when the sensor is directly over the line. Therefore, if the robot always follows that route which gives the highest electrical output from the sensor, it will always be following the line.

There is a problem with this technique: what does the robot do when the output from the sensor falls, indicating that it has left the line? With a single sensor system the best that the robot can do

Shaft Encoder



A shaft encoder allows a robot to know how far it has moved by measuring how much the axles of its wheels have rotated. The device is a calibrated disc, which is fitted on a shaft or axle. The circular plate is divided into a number of concentric rings, each marked out in areas that are either transparent or opaque. With a light source and a light sensor for each ring it is possible to work out the exact orientation of the shaft.

The accuracy of a shaft encoder depends on the number of rings there are on the disc. Our illustration shows a shaft encoder with three rings, which can encode the binary numbers 000 to111 (decimal 0 to 7). This encoder gives an accuracy of 360/8 = 45 degrees. Eight concentric rings would give an accuracy of 1.41 degrees

is to wander around until the output from the sensor rises again, showing that it is once more over the line. Then it can continue in the direction it is headed. This system is not quite as random as it may appear. For example, if the robot was going left when the output from the sensor dropped, then it makes sense that it would turn right in an attempt to find the line again. Also, having found the line, it is fair for the robot to asume that the direction it should now head in is somewhere between the (left) course it was following when it lost the line and the (right) course it had to follow in order to find it again.

A system that reduces the amount of time a 'derailed' robot has to spend finding the right direction again uses two sensors aimed at either side of the line. This means that when the robot is on the line, the output from both sensors is low. If the robot starts to wander off the line then the output from one sensor will rise. This means that the robot knows immediately that it has gone wrong and in which direction it has made its mistake. If the robot wandered to the right, the output from the left-hand sensor would rise, and the robot would take this as a signal to turn to the left, which would bring it back on course again.

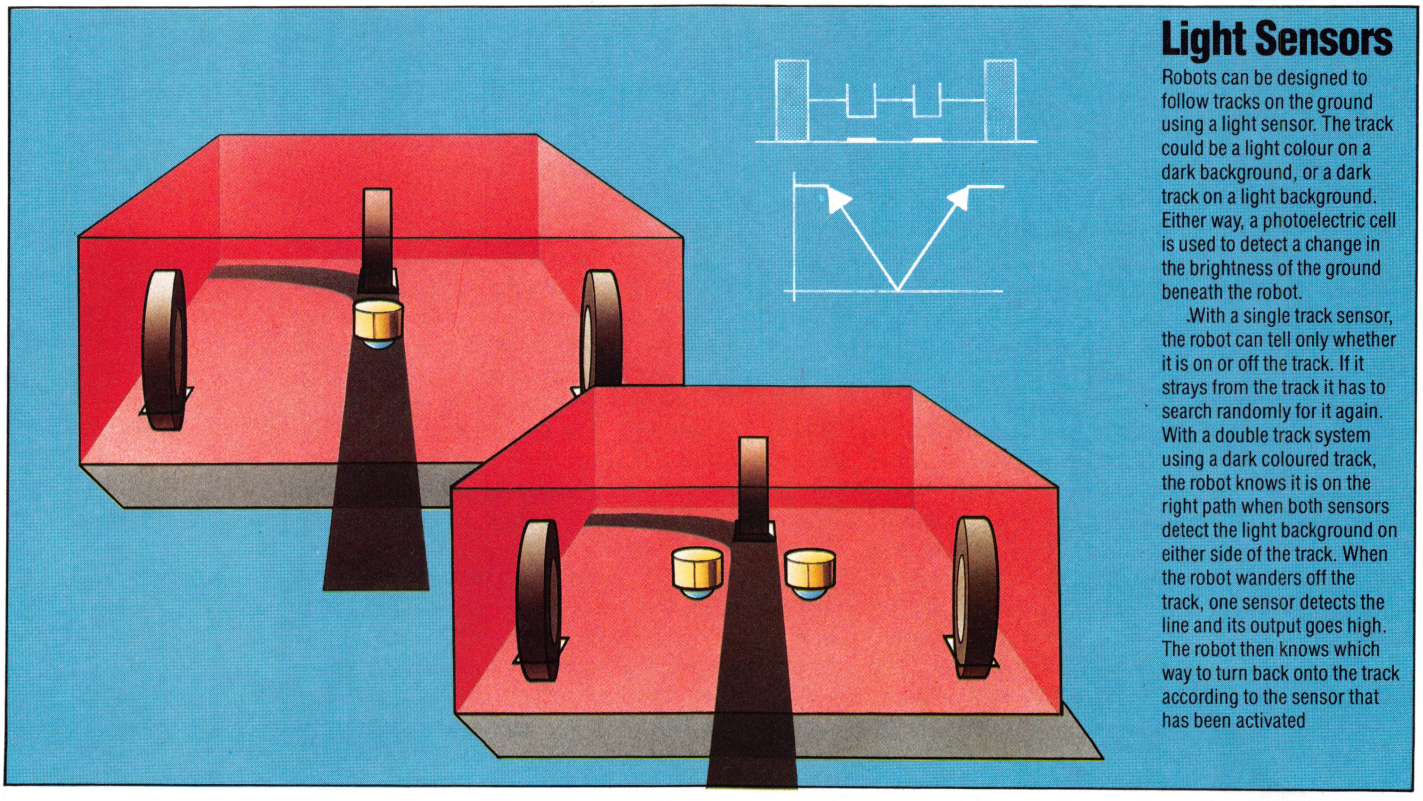
This system does not have to have a white line on a dark background — it would work equally as well with a dark line on a light background. What matters is the contrast — and that the programming tells the robot what to do when a sensor reads an incorrect value.

The other system used for track-following robots involves sending a small electric current along a wire placed in the floor. This current generates a small magnetic field around the wire, which is detected by a sensor. This need not be a complicated sensor — a small coil of wire will pick up the magnetic field and produce a small voltage that can then be amplified and will act in just the same way as the light sensors do. Industrial robots that need to move around often rely on a wire buried in the ground beneath them. If they relied on a line painted on the surface all would be well until the floor got dirty.

REMOTE CONTROL

Another method involves a human operator controlling the robot from a distance. This is particularly useful in circumstances where the tasks that the robot has to perform could be performed just as well by a human being but the environment is too hostile for this to be safe. Examples of this are bomb disposal, handling dangerous chemicals or radioactive materials and working in areas which are too hot, too cold or simply too dangerous for people to work in. A well-known robot of this type is the Russian Lunokhod 1, which was landed on the moon by Luna 16 in 1970. This was a robot on wheels that collected information from the surface of the Moon under the radio control of human beings back on Earth.

Controlling robots of this type is little different to controlling a radio controlled model aeroplane. The radio signal may be either an analogue signal, which varies in strength according to the amount by which the robot is required to move, or it may be a digital signal, which makes up a bit pattern giving details of the movements to be made. Analogue communications tend to be less successful than digital methods because other



KEVIN JONES

Light Sensors

Robots can be designed to follow tracks on the ground using a light sensor. The track could be a light colour on a dark background, or a dark track on a light background. Either way, a photoelectric cell is used to detect a change in the brightness of the ground beneath the robot.

With a single track sensor, the robot can tell only whether it is on or off the track. If it strays from the track it has to search randomly for it again. With a double track system using a dark coloured track, the robot knows it is on the right path when both sensors detect the light background on either side of the track. When the robot wanders off the track, one sensor detects the line and its output goes high. The robot then knows which way to turn back onto the track according to the sensor that has been activated

factors may interfere with the signal strength of an analogue transmission. Try listening to a distant radio station and notice how the reception varies according to the time of day and the weather conditions. The same sort of problems can affect robot communications.

Digital methods can have problems as well, especially when interference causes bits to be missed out or inserted where they shouldn't be. To avoid this, messages to the robot are often repeated, with the robot acting only after it has received an identical message several times.

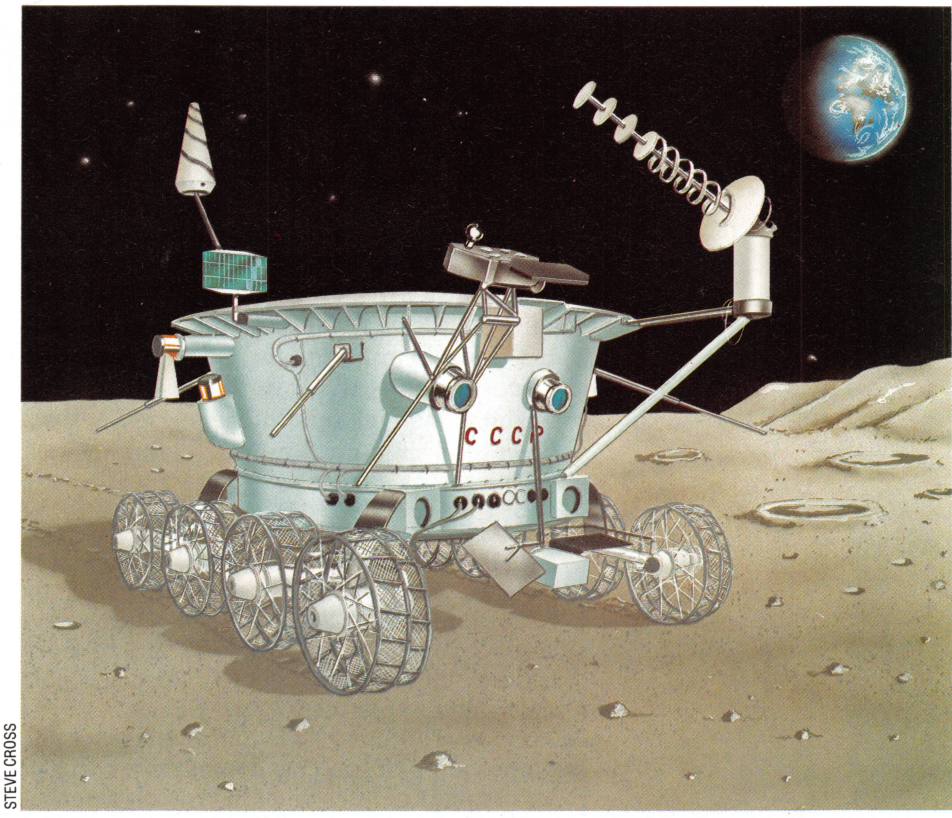
FEEDBACK SYSTEMS

A more sophisticated technique is to use a 'loop' system, in which the robot provides feedback to the transmitter concerning the signal it has just received. This could be regarded as a dialogue between the transmitter and the robot. For example, the transmitter might say 'move forward' and, having received this message, the robot says 'did you say move forward?', to which the transmitter replies 'yes', and the robot then moves forward. This can help to avoid serious mistakes if the robot is handling nuclear waste or is about to step into a crater on the Moon.

The same general techniques can be applied to other means of remote control. For instance, some robots can be controlled via infrared emitters of the sort used in remote control devices for television sets. Or they might be controlled by ultrasonic sound, rather like a dog whistle, or by audible sound of a distinctive nature, such as a series of hand claps. Whichever method is used, the underlying techniques of passing the message and making sure that the robot has received it remain the same.

If the human operator is fairly near the robot it may not be necessary to use such sophisticated techniques —the commands to the robot could be transmitted through a connecting wire. There is also the possibility of using more than one wire, which is equivalent to having several channels on a radio controlled aeroplane. But, in the case of the robot, the extra wires are usually used to provide parallel instead of serial communication (a string of bits is sent out in parallel along all the wires rather than as a series of pulses along one wire). This allows faster communications with the robot. Perhaps even more important is the fact that most computers have a parallel port on them. This provides a convenient way of communicating instructions to the robot from a computer keyboard.

If the robot movement is to be controlled by a human operator sitting at a computer keyboard, and the operator can see the robot, then there is little difference in principle between controlling the robot via a human operator and controlling the robot via a computer. This is because, like the radio controlled aeroplane, the operator can always see what the robot is doing and can correct any errors immediately. But if the robot is some distance away (on the Moon for instance, or even in the next room), or if the robot is to be controlled via a program within the computer rather than by real-time keyboard commands, then the robot must be slightly more intelligent.



STEVE CROSS

A Giant Leap For Robotkind
The USSR's Lunokhod 1 was landed on the Moon in 1970 to collect information about the nature of the surface and the atmosphere. It was not a true robot — being controlled by radio from Earth — but its indifference to lunar conditions enabled the spacecraft to carry a larger scientific payload than would have been possible with human passengers and their elaborate life-support systems.

Like all remote-controlled objects in space, Lunokhod suffered from the three-second lag between its transmitting information to Earth and receiving a control signal in reply

Essentially what is needed is some form of feedback. This is a process that enables the system to adjust what it is doing by reference to what it has done already and to what it should be doing. For instance, if you want a robot to travel three feet across the floor and you are controlling it directly, you can start it moving, judge its progress, and stop it when it has gone three feet. This is because you have visual feedback on the robot — you can see how far it has gone, how far you want it go, and you can correct its actions accordingly.

In the absence of human sensory feedback, the robot has to provide some of its own if it is to move accurately. The line-following robot uses feedback from the line it is following on the ground and, equally, the computer-controlled robot must use some feedback if it is to travel exactly three feet forwards. One of the most commonly used methods of providing the necessary feedback is a *shaft encoder* — a circular disc attached to the main axles of the robot's wheels, which gives a very precise measure of how far they have rotated. So, if the computer sends instructions to the robot to move forward three feet, the robot can start moving and, at the same time, monitor the signals coming from its shaft encoders to see how far the robot has moved. If the robot has to go further it can carry on moving. When it gets there it can stop, and if it should happen to overshoot its mark then it can always back up by the correct amount calculated from the information sent from the shaft encoders.

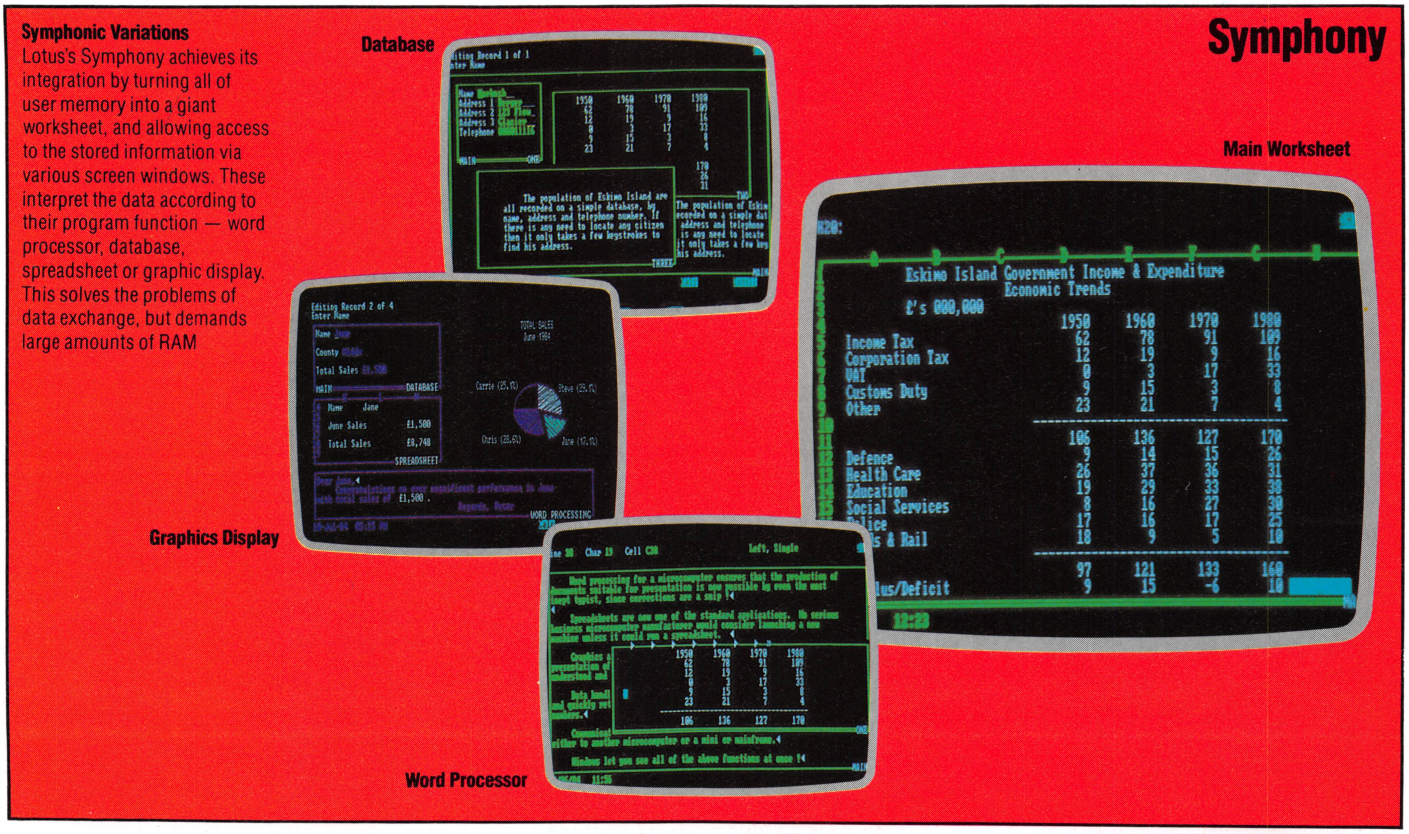
SYMPHONY IN SOFTWARE

In the first instalment of this series we considered the principles behind integrated software design. Now we look at Lotus's 1-2-3 and Symphony, and Psion's Xchange, three packages that are designed for large business systems but whose techniques will soon be applied to lower-priced machines.

As we have already seen, integrated software requires an environment in which the user has instant access to all the different tasks that may be required, where operating procedures remain the same no matter which application is being used, and where information may be moved freely between different applications. There are many different ways of achieving these aims.

Lotus 1-2-3 uses the familiar spreadsheet format, in which figures and formulae are entered into a matrix of 'cells' and can be freely amended and instantly recalculated. However, 1-2-3 offers many extra facilities and can be used for much more than just financial forecasting and analysis. The spreadsheet cells may be used to store information such as names and telephone numbers as well as numeric data, so a specific area of the grid may be used as a table containing relevant details — for example, a list of clients and their associated account numbers. As 1-2-3 offers functions for searching for and reorganising such information, this grid area may in effect be used as a small database. It is also possible to take a set of cells containing numeric data and use 1-2-3 to display this information in the form of different types of graph, thus removing the need for a separate business graphics program. Finally, 1-2-3's text-handling capabilities mean that it can be used for memo writing, although memory limitations preclude its use as a true word processor.

This combination of different facilities means that 1-2-3 is the only program that many users ever need. Because all the information for different applications is contained in a single spreadsheet, it is easy to achieve results that would be impossible with traditional programs. For example, let's assume that a 1-2-3 user operates several different newsstands in different parts of a large city, and needs to record weekly, monthly, quarterly and annual sales figures for each location. This is best done by placing the location of each stand and its sales figures into a spreadsheet. Formulae are written in such a way that the only figures that must be changed by the



user are the weekly receipts for each stand — other figures are then adjusted automatically.

So far, this is all standard spreadsheet material, but what if the owner wishes to put the stands in order of sales, so that the location with the highest sales is at the top of the list? These stands would initially be entered in alphabetical order, but will need re-sorting each week on receipt of the new sales figures. With Lotus 1-2-3 this may be done quickly and easily. The newsstand owner may require a weekly chart that shows how each stand has performed; a sequence of keypresses will allow this information to be retrieved for the spreadsheet/database, displayed in graph form, and printed out.

LOTUS SYMPHONY

Lotus's follow-up to 1-2-3 is called Symphony, and follows the same principle of basing applications on the spreadsheet format. However, Symphony allows the user to divide the screen display into separate windows, each of which focuses on a different part of the spreadsheet. Each window is formatted in a manner appropriate to the information it displays.

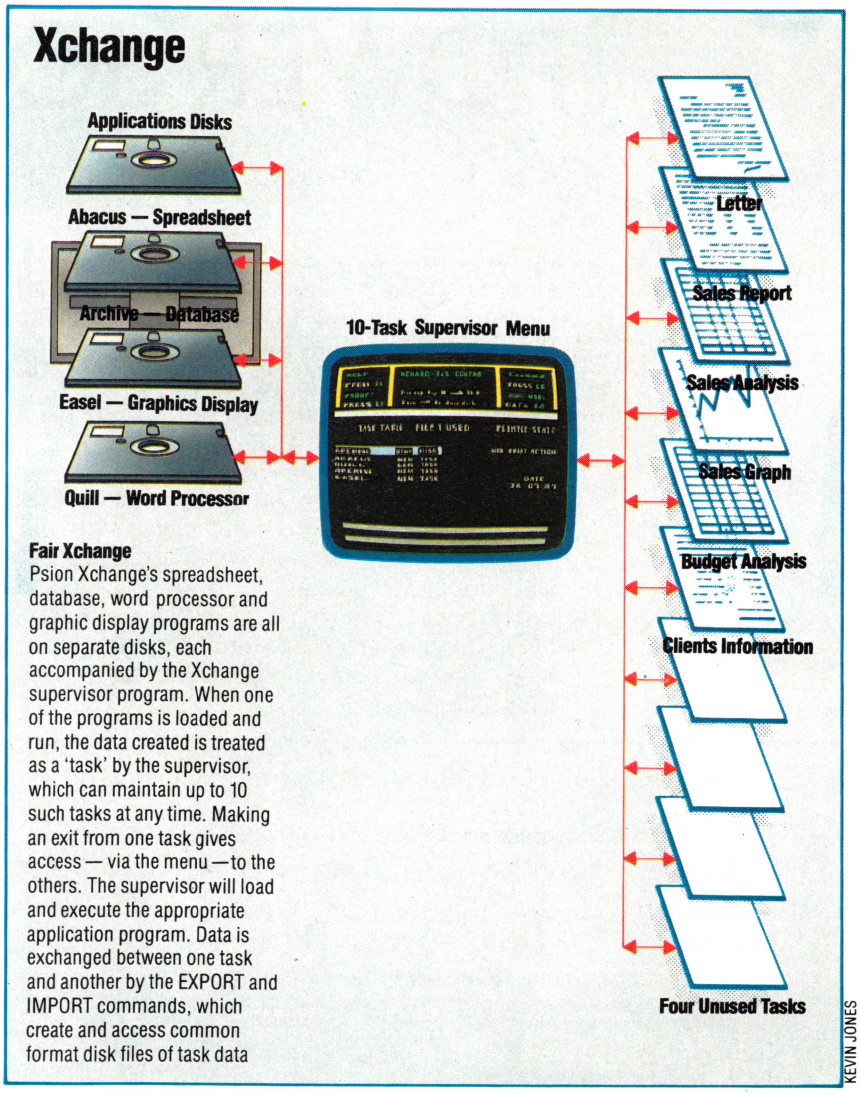
If the information to be displayed is held as text, the window takes the form of a small word processor screen, with margins and tab stops clearly marked. If a graph display is required, the window will show the labelled and scaled axes. Database information is displayed with each entry having its own screen; this looks like a card-index record. So, although Symphony is really an overgrown spreadsheet, it gives the impression of having four major applications all onscreen and working at the same time.

Like 1-2-3, Symphony can 'learn' particular sequences of keystrokes so that the user can automate any operations that are carried out frequently. The small programs that activate the sequence are called 'keyboard macros'. Symphony also includes its own high-level programming language. Programs are stored on the worksheet in the same way as all other data, and have access to all the operations available: so, if you have a task such as an invoicing or stock control system, you can write the program in Symphony's programming language and it will automatically be part of all the applications in the Symphony 'environment'. Once you are familiar with Symphony, you will find it easier to write programs in its command language than it is to use a separate programming language such as BASIC because Symphony already deals with such tasks as drawing graphs or searching for and organising data.

Symphony is just one of several similar systems that are now on the market. Ashton Tate's Framework is a strong competitor — this provides a similar range of functions but hides its underlying data structures to an even greater extent. Both Symphony and Framework are expensive (around £500 each) and require large amounts of memory. Symphony will work with 320 Kbytes of RAM but really requires 512 Kbytes to make the most of its facilities, while Framework needs a minimum of 256 Kbytes. As a result of these demands, the packages will run on 16-bit microcomputers only.

Interestingly, neither Symphony nor Framework requires information or portions of program to be swapped between disks and main memory, as is the case with most business programs. In theory, of course, computer memory continues to become cheaper and cheaper, so it is not unreasonable for software developers to assume that most users will have large amounts available. In practice, however, this is not yet the case and it will be some time before such memory-intensive integration becomes commonplace. Although a program such as Symphony sets new standards of performance, such software is still constrained by hardware limitations — it's only by being such a large, carefully crafted program that Symphony manages the things it does.

An alternative method of providing integrated software has been developed over the last 20 years, and packages that use this method are now starting to appear on the market. In the next instalment we will consider possible future developments in integrated software.

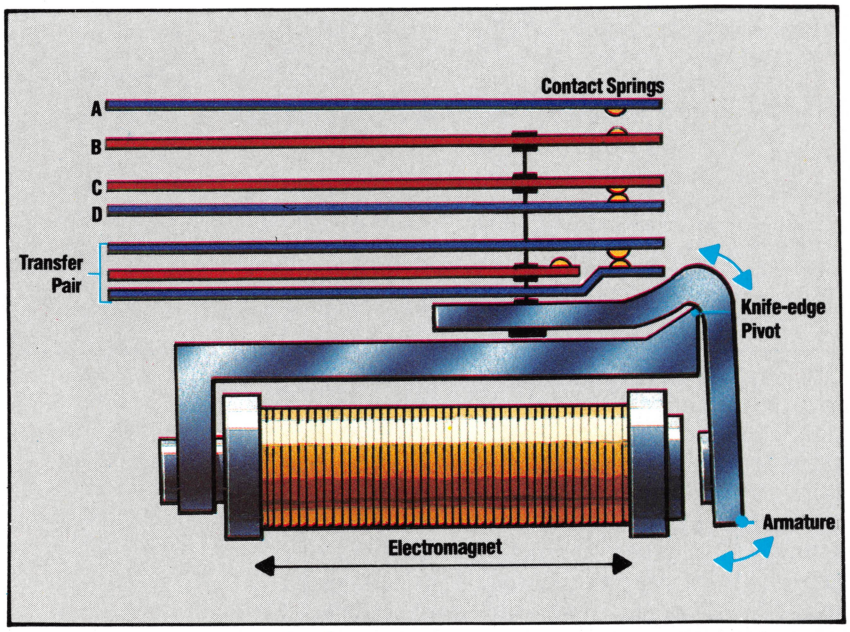


Fair Xchange
Psion Xchange's spreadsheet, database, word processor and graphic display programs are all on separate disks, each accompanied by the Xchange supervisor program. When one of the programs is loaded and run, the data created is treated as a 'task' by the supervisor, which can maintain up to 10 such tasks at any time. Making an exit from one task gives access — via the menu — to the others. The supervisor will load and execute the appropriate application program. Data is exchanged between one task and another by the EXPORT and IMPORT commands, which create and access common format disk files of task data

POWER SOURCE

Our Workshop series continues with an explanation of how to build a mains relay box. Using this, your computer will be able to switch the house lights on and off at preset or random intervals, and can be used to program a video or audio recorder.

Electrical relays are on/off switches that can be activated by an electrical signal. In our application, relays are used to switch high voltage and current appliances using a low voltage and current signal. Many types of relay are available, but the most common is the armature-type, which relies on a solenoid to make and break connections.



The relay makes and breaks contacts under the action of small movements in the armature. An appropriate voltage applied to the solenoid coil generates a magnetic field that attracts the armature. As the armature swings in towards the coil the spring contacts attached to the other end of the armature are made to move vertically upwards.

The arrangement shown is in the 'non-energised' position; that is, with no voltage applied to the solenoid. In this position, the contact pair AB is open and the pair CD is closed. When the solenoid is energised, springs B and C move upwards, causing A and B to close and C and D to open. This arrangement can be used in one of two ways: either to switch in one circuit whilst switching out another, or, more simply, to complete or break a circuit.

In addition to this mode of operation, a relay can also act as a transfer mechanism. In the diagram the lower three springs are arranged so that — in the non-energised position — the top and bottom springs are in contact. When the solenoid is energised, the middle spring moves up and makes contact with the top spring, thus breaking the contact between the top and bottom springs.

Parts List

Quantity	Item	Maplin Number
2	10A 240V contact 8-15V relay	YX97F
2	Single unswitched mains socket	HL68Y
2	Single 29mm pattress	YB15R
2 metres	6 amp 3-core mains cable	XR03D
2	Mains plug, 13 amp fused	RW67X
*	4mm plugs	*
*	0.5 metre 2-way ribbon cable	*
*	Small pieces of veroboard	*

NB: Items marked * should be left over from previous projects (see page 524). These quantities will provide two single-socket mains relay boxes. The output box can drive four such boxes — if so desired these may be double- or even triple-socket boxes; construction principles are exactly the same, only the parts differ.

Inside The Box

Check all connections for security and continuity, and inspect the board once again for track bridging. Ensure that there is no electrical path between the mains lead and the signal lines.

Glue the board into a corner of the pattress using an epoxy resin glue. Some makes of all-purpose household glues conduct electricity, so avoid these at all costs. If you are unsure of the conducting properties of your chosen adhesive, try spreading a thin strip of glue on a piece of card, allow it to dry, and then connect a multimeter to each end: if the meter gives a reading, use a different glue!

Once the glue has dried, screw the lid on the box and put the 4mm plugs on the signal lines (these can be the same colour, as the relay will work despite the direction of the current). Now connect the 13 amp plug to the mains lead. The relay is rated at 10 amps, but for safety you should probably not switch any more than 5 amps through it, so put a 5 amp fuse in the plug: this allows you to control appliances rated at up to 1.2 kW.

Warning!

This is a very simple project, but anything involving mains power demands care and respect.

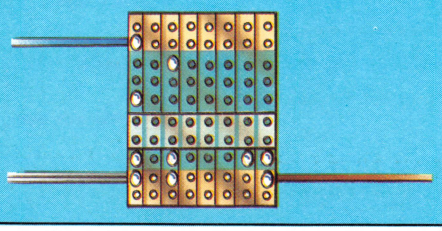
- **Disconnect all power sources before you work on any part of the box.**
- **Check all connections and insulations with a multimeter before applying power for the first time.**
- **Take care, and avoid all short-cuts. Remember — MAINS POWER CAN KILL!**

The Circuit Board

Cut the board to the shape shown, so that it will fit snugly into one corner of the pattress box. Make the track cuts, and solder in the relay as shown in the diagram.

Check the board very carefully before you go any further. Use the multimeter to check for bridging between tracks — a mistake here could kill!

Solder the brown mains lead and the two-way ribbon cable into place on the board. Remove one of the pre-formed slots in the pattress to accept the wires; but tie a knot in these before threading them through — the knot will prevent an accidental pull on the wires from damaging the board. Solder a short length of insulated mains conductor to the board, and connect this to the 'live' screw terminal on the socket. Connect the blue and yellow/green mains leads to the neutral and earth terminals respectively



Test Program

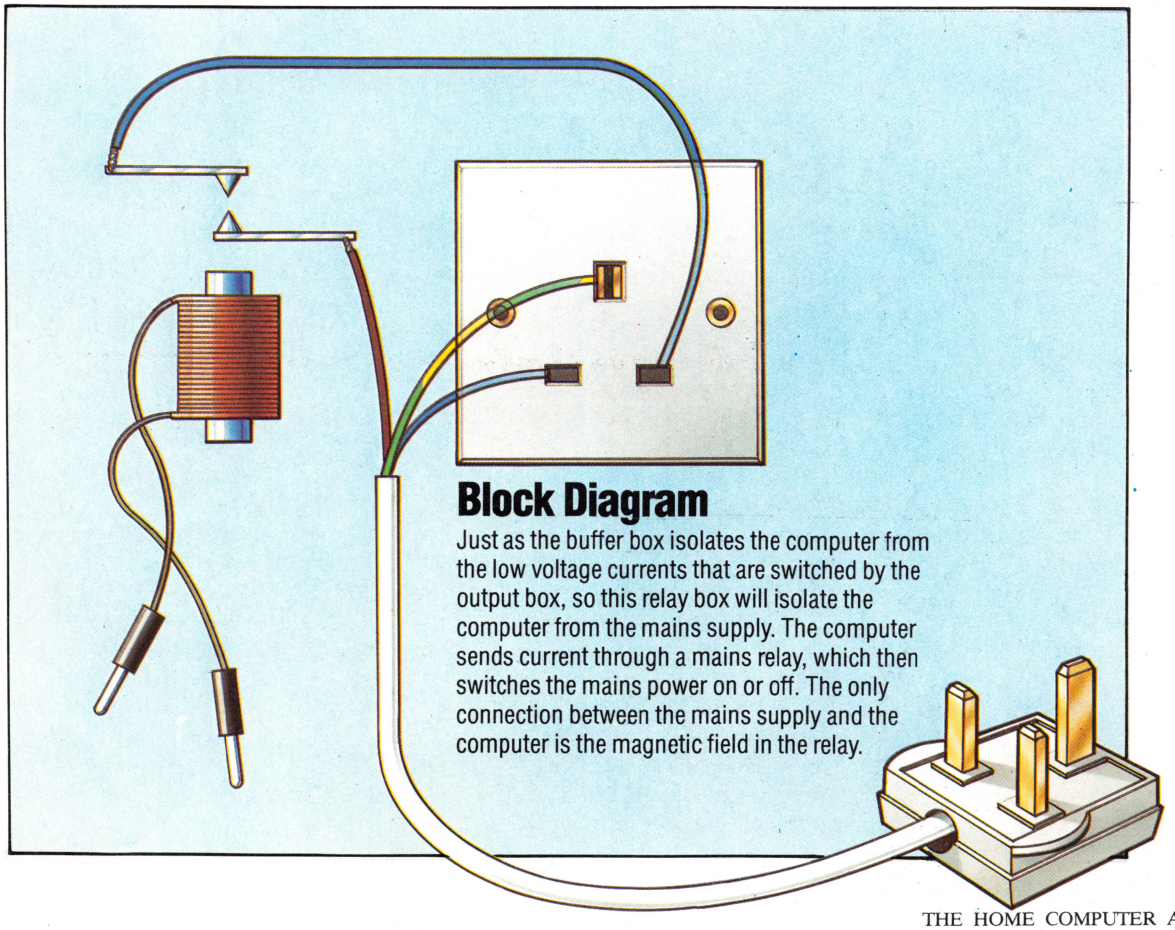
Once we have built the relay box and checked all connections thoroughly, we can test its operation by writing a short program to switch a mains-powered device on and off. A suitable device for such a purpose is a simple table lamp. This should be plugged into the mains power socket on the relay box, and the signal wires connected to the positive and negative terminals on line 0 of the low-voltage output box. The signal wires may be connected to either of the terminals without affecting the operation of the relay. The mains lead from the relay box is then plugged into a wall socket.

Once all connections have been made, type in the following short program; this switches the lamp on for five seconds and then switches it off again.

```
10 REM TEST MAINS RELAY
20 DDR=&FE62:DATREG=&FE60
30 ?DDR=255: REM ALL OUTPUT
40 ?DATREG=1: REM TURN LIGHT ON
50 TIME=0: REM SET TIMER
60 REPEAT
70 UNTIL TIME>500
80 ?DATREG=0: REM TURN LIGHT OFF
```

```
10 REM CBM 64 TEST MAINS RELAY
20 DDR=56579: DATREG=56577
30 POKE DDR,255: REM ALL OUTPUT
40 POKE DATREG,1: REM TURN LIGHT ON
50 T=TI: REM SET TIMER
60 IF (TI-T)<300 THEN 60
70 POKE DATREG,0: REM TURN LIGHT OFF
```

If, after running the program, the lamp fails to light, unplug the mains relay box before testing the connections.



Block Diagram

Just as the buffer box isolates the computer from the low voltage currents that are switched by the output box, so this relay box will isolate the computer from the mains supply. The computer sends current through a mains relay, which then switches the mains power on or off. The only connection between the mains supply and the computer is the magnetic field in the relay.

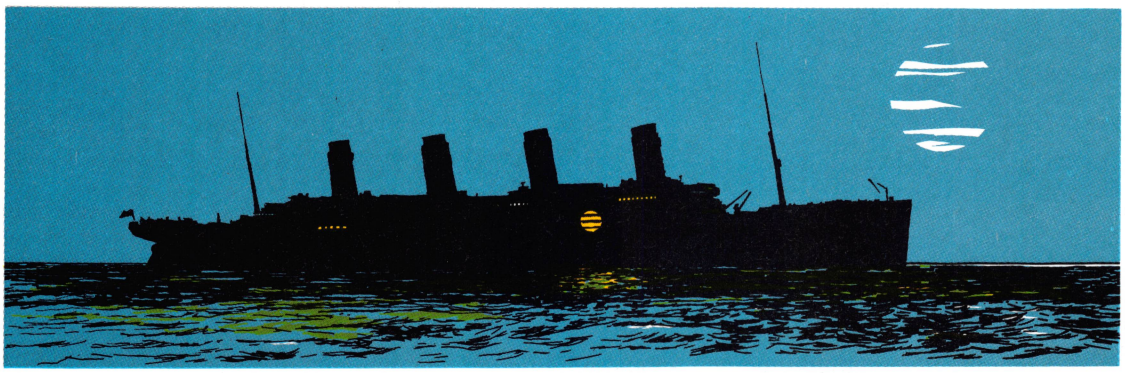
Morse Code Project

```
10 REM************************
11 REM*  C64 MORSE PRACTICE   *
12 REM*   PLUG A LAMP INTO    *
13 REM*   THE MAINS RELAY:    *
14 REM*   ENTER ANY ALPHA     *
15 REM*    STRING, AND IT     *
16 REM*   WILL BE FLASHED     *
17 REM*  AND BEEPED IN MORSE  *
20 REM************************
100 GOSUB 2000:          REM  INIT
150 FOR L=0 TO 1 STEP 0
200 PRINT"ENTER YOUR MESSAGE"
220 PRINT"TYPE 'BYE' TO QUIT"
240 INPUT"MESSAGE ";MS$
300 ML=LEN(MS$):M$=""
320 FOR K=1 TO ML
330 C$=MID$(MS$,K,1)
340 IF C$=>"A"ANDC$<="Z"THEN M$=M$+C$
350 IF C$=" " THEN M$=M$+C$
360 NEXT K:IF M$="" THEN NEXT L
400 ML=LEN(M$)
420 FOR K=1 TO ML
440 CH$=MID$(M$,K,1):CH=ASC(CH$)-64
450 IF CH=-32 THEN FORPP=1TO6*DE:NEXTPP
460 IF CH<>-32 THEN GOSUB 3000
480 FOR PP=1 TO 3*DE:NEXT PP
500 NEXT K
550 IF M$="BYE" THEN L=1
600 NEXT L
900 END
2000 REM********INIT******************
2100 DIM M$(26)
2110 DDR=56579:DATRG=56577:POKE DDR,255
2120 DE=25:RX=2*DE
2130 V=54296:LF=54272:HF=54273:W=54276
2140 A=54277:S=54278
2150 FOR K=LF TO LF+24:POKEK,0:NEXT K
2160 POKEA,24:POKES,129:POKEV,15
2200 DATA ".-","-...","-.-.","-..","."
2220 DATA "..-.","--.",",....","."
2240 DATA ".---","-.-","-.--","--"
2260 DATA "-.","---",".--.","--.-"
2280 DATA ".-.",",...","-","..-",",...-"
2300 DATA ".--","-..-","-.--",",--.."
2400 FOR K=1 TO 26:READ M$(K):NEXT K
2990 RETURN
3000 REM*********FLASH & BEEP********
3050 PRINT CH$,M$(CH)
3100 N=LEN(M$(CH))
3200 FOR C=1 TO N
3220 D=DE-(ASC(MID$(M$(CH),C,1))-46)*RX
3240 POKE DATRG,1            :REM FLASH
3250 POKE LF,172:POKE HF,57:REM BEEP
3260 POKE W,33:FOR PP=1 TO D:NEXT PP
3270 POKE W,32
3280 POKE LF,0:POKE HF,0    :REM UNBEEP
3290 POKE DATRG,0            :REM UNFLASH
3300 FOR PP=1 TO DE:NEXT PP
3320 NEXT C
3490 RETURN
```

```
 10 REM************************
 11 REM*  BBC MORSE PRACTICE   *
 12 REM*   PLUG A LAMP INTO    *
 13 REM*   THE MAINS RELAY:    *
 14 REM*   ENTER ANY ALPHA     *
 15 REM*    STRING, AND IT     *
 16 REM*   WILL BE FLASHED     *
 17 REM*  AND BEEPED IN MORSE  *
 20 REM************************
100 PROCinitialise
120 ALLOVER=FALSE
150 REPEAT
200 PRINT"ENTER YOUR MESSAGE"
220 PRINT"TYPE 'BYE' TO QUIT"
240 INPUT"MESSAGE",MS$
300 ML=LEN(MS$):M$=""
320 FOR K=1 TO ML
330 C$=MID$(MS$,K,1)
340 IF C$>="A"ANDC$<="Z"THEN M$=M$+C$
350 IF C$=" " THEN M$=M$+C$
360 NEXT K:IF M$="" THEN UNTIL FALSE
400 ML=LEN(M$)
420 FOR K=1 TO ML
440 CH$=MID$(M$,K,1):CH=ASC(CH$)-64
450 IF CH=-32 THEN PROCdelay(6*DE*IX)
460 IF CH<>-32 THEN PROCbeepflash
480 PROCdelay(3*DE)
500 NEXT K
550 IF M$="BYE" THEN ALLOVER=TRUE
600 UNTIL ALLOVER
900 END
2000 REM**********INIT*************
2050 DEFPROCinitialise
2100 DIM M$(26)
2110 DDR=&FE62:DATREG=&FE60:?DDR=255
2120 DE=3:RX=2*DE:IX=30
2200 DATA ".-","-...","-.-.","-..","."
2220 DATA "..-.","--.",",....","."
2240 DATA ".---","-.-","-.--","--"
2260 DATA "-.","---",".--.","--.-"
2280 DATA ".-.",",...","-","..-",",...-"
2300 DATA ".--","-..-","-.--",",--.."
2400 FOR K=1 TO 26:READ M$(K):NEXT K
2990 ENDPROC
3000 REM*****FLASHANDBEEP**********
3020 DEFPROCbeepflash
3050 PRINT CH$,M$(CH)
3100 N=LEN(M$(CH))
3200 FOR C=1 TO N
3220 D=DE-(ASC(MID$(M$(CH),C,1))-46)*RX
3240 ?DATREG=1            :REM  FLASH
3260 SOUND 1,-15,200,D    :REM  BEEP
3270 PROCdelay(IX*D)
3290 ?DATREG=0            :REM  UNFLASH
3300 PROCdelay(DE*IX)
3320 NEXT C
3490 ENDPROC
4000 REM*******DELAY***************
4100 DEFPROCdelay(time)
4200 FOR DD=1 TO time:NEXT DD
4490 ENDPROC
```

Morse Code

A	.–
B	–...
C	–.–.
D	–..
E	.
F	..–.
G	––.
H	....
I	..
J	.–––
K	–.–
L	.–..
M	––
N	–.
O	–––
P	.––.
Q	––.–
R	.–.
S	...
T	–
U	..–
V	...–
W	.––
X	–..–
Y	–.––
Z	––..
.	.–.–.–
,	––..––



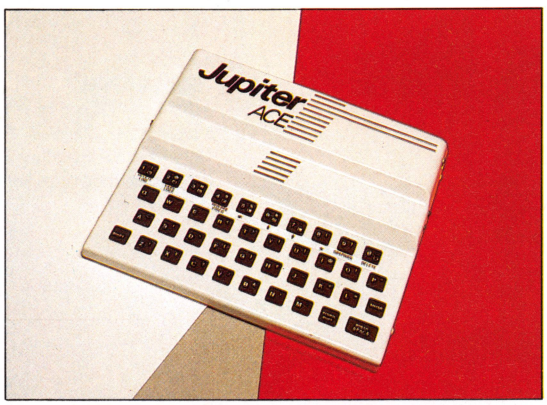
F

FORTH

FORTH was invented by astronomer Charles Moore in 1972 when he became dissatisfied with FORTRAN as a language in which to write telescope control programs. The specialised functions that he needed were difficult to write in FORTRAN because its structure and processes were too strongly oriented towards that language's scientific/mathematical purposes. Accordingly, he designed FORTH as a dictionary of *primitives*— the elementary functions of the language — and an editor/compiler/interpreter.

The editor is used to define new functions as expressions (or 'subroutines') created from the existing dictionary; the new functions are named and compiled, and thus added to the language. A function can be executed at any time through the interpreter by simply issuing its name as a command. FORTH treats all of user memory as one big Last In First Out (LIFO) stack, while program memory is a series of independent stacks (one per function) of machine code subroutine addresses. A function is executed by jumping to the first address on its program stack, popping any needed values off the main stack, pushing any results back onto the stack, and exiting to the next stacked address, until execution is complete. Arithmetic expressions are, therefore, written in *reverse Polish* — or *postfix* — *notation* (operands are grouped together and followed by their operators; thus `A+B*C` is written `B C A * +`) because this is a stacked-oriented notation.

Programming in FORTH, then, really consists of developing a customised version of the language to suit each application. The language's chief virtues are its 'extensibility' and speed of operation. Because of its extensibility and because it brings the user closer to the computer's operations than is normal with high-level languages, FORTH has been acclaimed as a replacement for BASIC but, although it is available in various versions for most micros, only one — the now-defunct Jupiter Ace — has been manufactured with FORTH rather than BASIC as its resident language.



FORTRAN

Developed by IBM in 1956, FORTRAN (derived from FORmula TRANslation) was the first commercially available high-level language. It had two main purposes: to demonstrate that high-level, quasi-English programming languages could be compiled quickly and efficiently, and to make computers more generally accessible to scientists and engineers who might be prepared to learn a language rather like the algebraic expressions in which they formulated their ideas, but who had neither the time nor the patience to learn machine code. In both of these aims FORTRAN has been enormously successful; it is still the most widely used of the high-level languages, and a new version is due in the late 1980s. Several versions are also available for microcomputers.

An important early development was the ability to create system libraries of independently-compiled FORTRAN subroutines: all mainframe systems have such libraries, and so important a resource are they that other languages — such as PASCAL — are configured so that they can call FORTRAN routines from the libraries.

FORTRAN'S legacy is the group of languages descending from it—chiefly ALGOL, PASCAL and BASIC—but its true historical significance is probably that it enabled computers to move out of university computing laboratories and into the classrooms and workshops, where they could become taken for granted as everyday scientific equipment. From there it was a short step into offices and homes. FORTRAN brought computing within the reach of the non-specialist, and was perhaps, therefore, the first step on the road to user-friendliness.

FOURTH GENERATION

A generation in the development of computers seems to span about ten years, and begins with the development of an expensive new technology which is commonplace by the end of that period. The first generation began in the late 1940s with the first stored-program thermionic valve machines; the second generation machines appeared in the late 1950s and used discrete transistor logic; the third generation — typified by the IBM 360 — began in the early 1960s with integrated families of machines and comprehensive operating systems; and the *fourth generation* appeared in the early 1970s with the introduction of Large Scale and Very Large Scale Integration (LSI and VLSI) chip circuitry. As such, it includes mainframe, mini and microcomputers. Micros have themselves gone through several stages of development to arrive in the middle 1980s as credible small-scale computers, supporting fourth generation features such as large memories (one megabyte or more), networking, multi-tasking and integrated software.

The fifth generation is expected to appear in the late 1980s. Its characteristic features are likely to be natural-language programming, speech recognition and generation, and a degree of artificial intelligence in its operating systems and applications software; it will probably be developed in Japan.

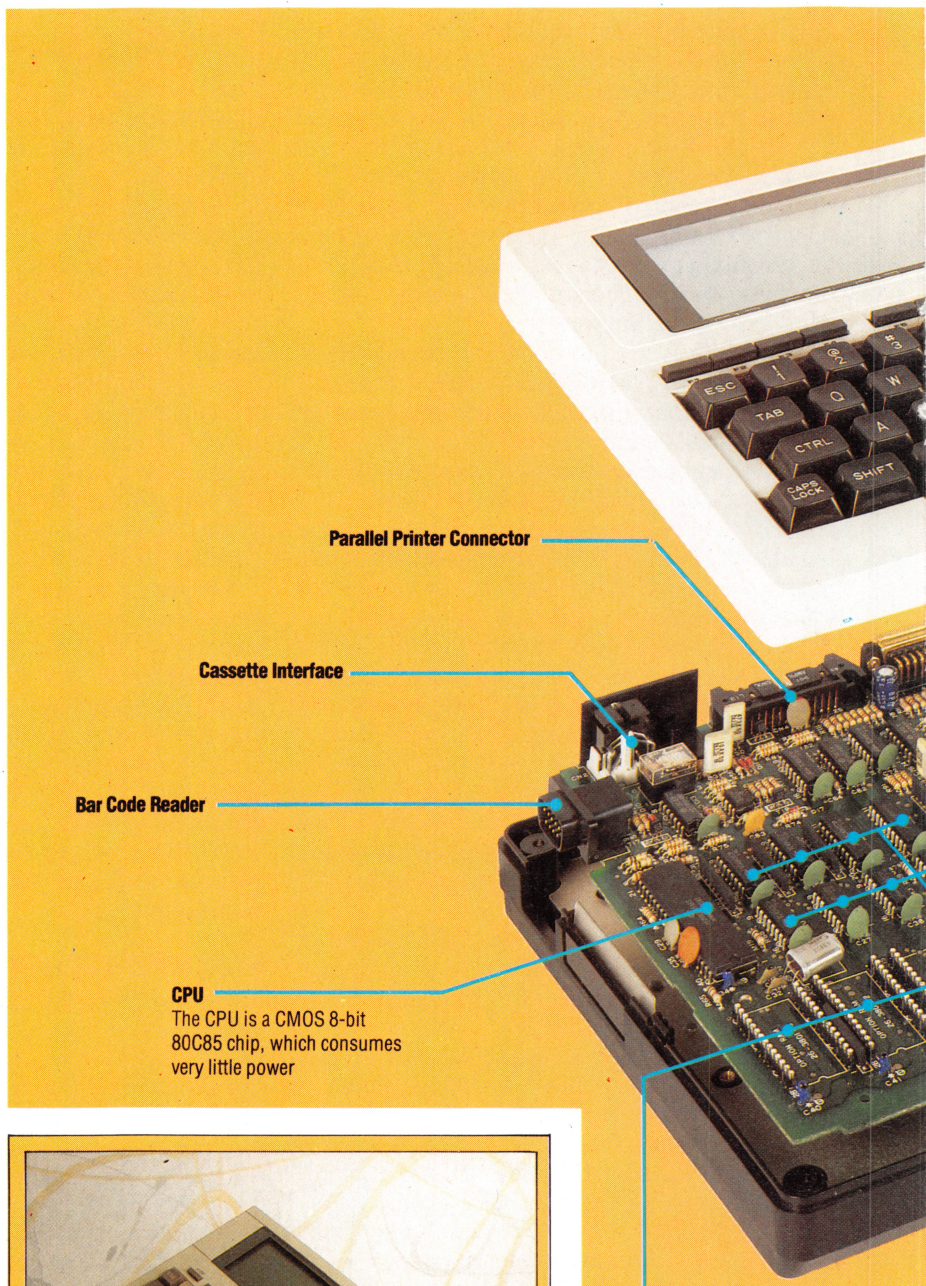
THREE OF A KIND

As a result of the ever-growing demand for 'computing on the move', manufacturers are concentrating on the lucrative portable computer market. Here, we take a look at one of these 'lap-held' machines — the Tandy Model 100 — and compare it with two, essentially similar, competitors.

The process whereby a manufacturer buys a completed product, changes a few elements to make it look unique, then repackages it as a 'custom manufactured' item, is known as 'badge engineering'. This technique has existed for a long time in the consumer electronics field, with products such as televisions and hi-fi equipment. The same technique is now being used in the computer market, and three popular portable computers — the Tandy Model 100, the NEC PC8201A and the Olivetti M10 — are the result of just such an arrangement. All three machines are manufactured by the same company, the Japanese Kyocera firm, and are sold to Tandy, NEC and Olivetti, who package the machines and market them under their own labels. Here, we consider the Tandy Model 100, and highlight the differences between this machine and its siblings.

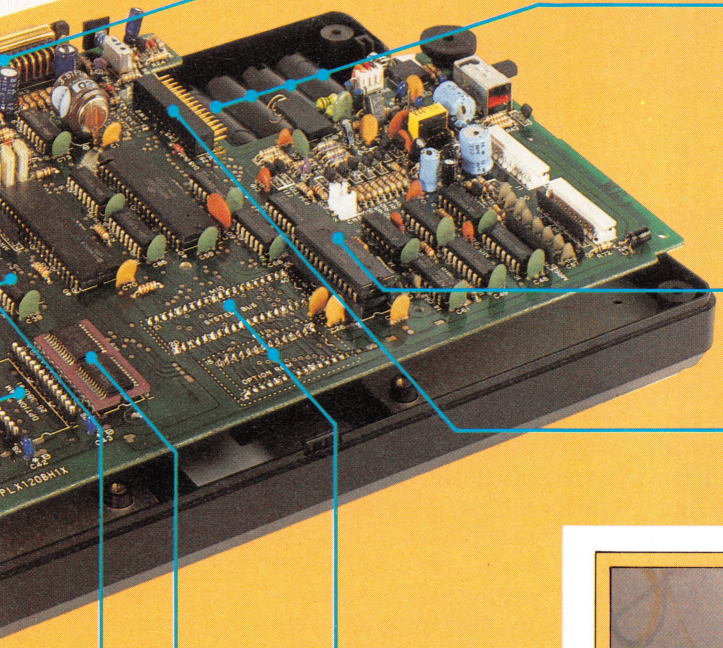
Weighing slightly less than 1.8 kg (4 lb), the Tandy, NEC and Olivetti models fall comfortably into the 'lap-held' category. The Model 100 has a full QWERTY-style keyboard, built-in ROM-based software and a battery-operated LCD screen. It can be run entirely on battery power and the contents of RAM are not lost when the machine is switched off. Files may be stored in RAM and accessed directly as if the memory were a cassette or disk. The Model 100 may also be connected to a cassette or disk drive for external storage, but the permanent memory makes it easy to store important data 'on the run'.

The LCD screen provides eight lines of 40 characters, and has the ability to mix text and graphics. The display is composed of 15,360 dots, each of which may be addressed individually. Characters are formed in a 6 by 8 matrix, and upper- and lower-case characters may be displayed. The Model 100 features a full international character set, as well as a special set of graphics characters, unlike the NEC machine, which has only three graphics characters. Both the NEC and Tandy models have LCD screens that lie flat in their cases, but the Olivetti M10 features a movable screen that can be tilted to a comfortable working angle, thus providing extra flexibility. The NEC and Tandy screens have adjustable contrast controls to improve screen clarity.



NEC PC8201A
Although the NEC PC8201A is exactly the same size as its siblings, this machine has a significantly different keyboard. The cursor keys have been moved out into a small cluster, the function keys have been reduced from 8 to 5, and the keyboard layout is slightly different. In addition, the NEC has only three of the standard programs in ROM: Text, Schedule and Telecom

Modem Connector
This is a standard RS232 serial communications port. Telecommunications software is built into the machine



Power Supply
The Model 100 runs for up to 20 hours on 4 'AA' alkaline batteries. Internal memory is maintained for up to 30 days by rechargeable nickel-cadmium batteries, which are automatically recharged when the power is on

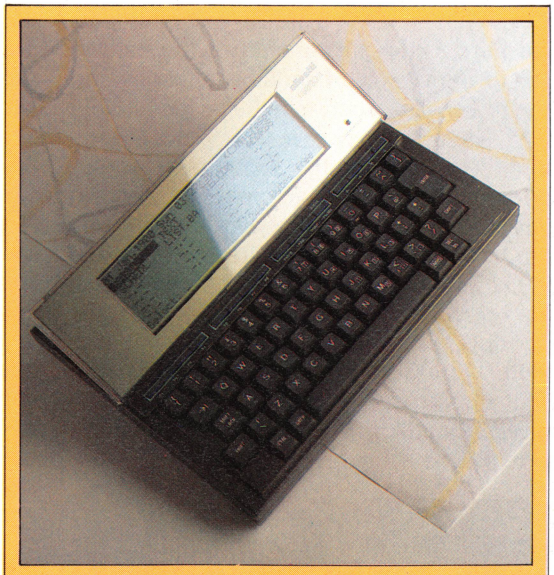
Standard ROM
This chip contains the built-in Microsoft BASIC and software

LCD Connector
A ribbon cable connects the LCD screen to the system board here

System Bus and ROM Slots
These empty slots are for future expansion of the system's ROM, and input/output control

Standard 8K RAM

Keyboard Unit
These chips control the input and output of the keyboard, and contain the character sets



Olivetti M10
The Olivetti version of this machine has one interesting touch unique to itself: the LCD screen can tilt up to an angle of about 40° — making the display somewhat easier to read. It has essentially the same keyboard as the Tandy 100, however, and all five standard ROM-based software packages

TANDY MODEL 100

PRICE
£449 inc VAT

DIMENSIONS
300×215×50mm

CPU
8-bit 80C85 CMOS, 2.4 MHz

MEMORY
8K or 24K RAM, expandable in 8K increments to a total of 32K; 32K ROM including software and Microsoft BASIC

SCREEN
LCD 40 columns × 8 lines; 240 × 64 dot-addressable graphics; ASCII and international characters, 39 graphics characters

INTERFACES
Parallel printer, cassette, RS232 serial port, bar-code reader, system bus

LANGUAGES AVAILABLE
Microsoft BASIC

KEYBOARD
Standard typewriter-style 56-key keyboard; embedded numeric pad; 8 programmable function keys; 4 command keys and 4 cursor control keys

DOCUMENTATION
48-page BASIC quick reference guide; 200-page detailed operations manual

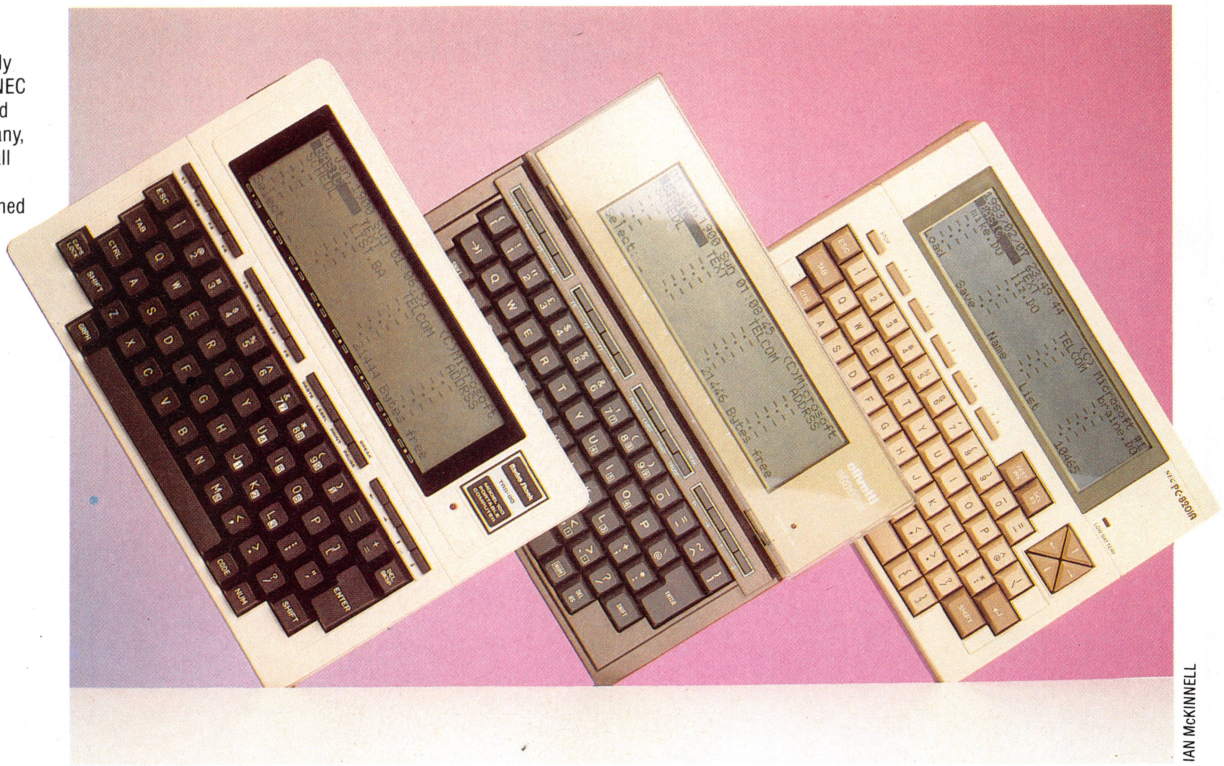
STRENGTHS
The Tandy 100 is small yet has most of the features needed for 'serious' computing. Permanent memory and battery operation make it very portable

WEAKNESSES
RAM limitations reduce the practical use of the machine to portable applications only. It would not have the ability to 'transcend' portable status and function as a full desktop computer

DIFFERENCES
Olivetti M10:
57-key keyboard; 47 graphics characters; tilting screen unit; one user manual
NEC PC8201A:
57 keys; cursor rose; 5 function keys; 16K RAM expandable to 96K; three graphics characters only

NEC COURTESY OF CL PROJECTS IAN McKINNELL

Chips Off The Old Block
Although distributed by three different companies, the Tandy Model 100, Olivetti M10 and NEC PC8201A are all manufactured by the same Japanese company, Kyocera. There are some small differences between the three machines, but even an untrained eye can see that the three portable computers share a common heritage



IAN McKINNELL

The Tandy's high-quality keyboard features special keys to access the built-in graphics or to change several of the letter keys into a numeric keypad. Using this facility, key M becomes 0; J, K and L become 1, 2 and 3; U, I and O become 4,5 and 6; and 7, 8 and 9 retain their normal function. All three machines have four cursor keys, but the position of these varies. The Tandy and Olivetti models have four small keys side by side, located above and to the right of the regular keyboard; the NEC PC8210A has a cursor pad, with the four cursor keys forming a square.

The machines also feature programmable function keys, which are used with the built-in software to manage file-handling functions and movement within and between the programs held in ROM. Again, there are differences here. Tandy's Model 100 has eight function keys, plus four additional keys that are used to perform internal tasks. PASTE is used to move data from one program to another; LABEL assigns names to the function keys so the user always knows what each function key does; PRINT sends files directly to the printer; and the BREAK key halts program execution. This layout is repeated on the Olivetti M10, but the NEC PC8201A has five function keys, programmable for a total of 10 functions, and a Pause key.

MEMORY CAPACITIES

The Model 100 and the M10 are supplied with either eight Kbytes or 24 Kbytes of RAM and this can be expanded to 32 Kbytes with the addition of an internal RAM pack. The NEC is slightly different: this is supplied with 16 Kbytes, but may be expanded to 64 Kbytes internally, or 96 Kbytes if the built-in expansion port is utilised.

The Model 100 comes with Microsoft BASIC and a small 'housekeeping' system that manages the internal software. On power-up, the files stored in memory are displayed, along with the titles of the supplied internal software programs.

Supplied programs include Text, a small word processor that is suitable for drafting memos or writing letters or short reports; this is especially suitable for note-taking, and should be a boon to journalists, students or business users. Schedule is a small database program, specifically designed to help you keep track of appointments, expenses, 'things to do' and other reminders. A built-in search function makes it easy to find information quickly. A third program, called Address, is a similar small database and appears unnecessary as Schedule is available. Finally, there is an RS232-based communications program called Telecom, which allows the Model 100 to be connected to a modem for telephone communications — with a few keystrokes, data can be sent to or received from remote computers. The NEC PC8201A comes with only BASIC, Text and Telecom.

All three machines are well equipped with interfaces, each possessing an RS232 communications port, a parallel printer port, cassette interface, and a socket for a bar-code reader. The Tandy and Olivetti models include a system bus, while the NEC adds two extra serial ports to its list of interfaces.

The use of one basic machine, with slight differences between the three different models, has meant that the manufacturers can provide high-quality products without any one company having to shoulder the full development costs. At a cost of around £450 for the basic versions, all three of these lap-helds offer good value for money.

WHICH BIKE?

In the last instalment we gave you a BASIC program for a motorcycle game on the ZX Spectrum (see page 632). Here we provide versions of the same game for two other machines — the Commodore 64 and the BBC Micro.

Unlike the versions of BASIC used by the Spectrum and BBC, Commodore 64 BASIC doesn't have any commands that allow us to plot individual pixels. In the version of the game we give here, we use low resolution characters to draw the path of the 'light cycles'. A reverse-field space character, with POKE code 160, is used: to plot this character to the screen we have to POKE this value to the screen map in memory and specify the colour in the corresponding location in the colour map.

Like the Spectrum version, the Commodore game is unstructured for maximum speed of execution. At those points in the game where speed is unimportant, such as after a collision, some structuring is introduced in the way of subroutine calls to increment the score and flash the screen.

Because BBC BASIC runs considerably faster than Spectrum or Commodore BASIC, and allows structured modules to be called as procedures, the BBC version of the game is written in a highly structured way. Most versions of BASIC allow structuring by using subroutines, but this slows down execution speed because a search must be made each time a subroutine is called. BBC BASIC, however, makes a note of the location of a procedure when it is first called, and stores this in a reference table.

Commodore 64

```
10 REM CBM 64
15 POKE53281,0:POKE53280,4:REM SCR/BORD COLOUR
20 SC=1024:CO=55296
25 PRINTCHR$(147):REM CLEAR SCREEN
30 X1=2:Y1=12:X2=37:Y2=12
35 DX=1:DM=-1:DN=0:DY=0
40 PRINTCHR$(19)CHR$(158)"PLAYER 1:"S1
50 PRINTCHR$(19)TAB(27)"PLAYER 2:"S2
52 FORI=1TO8:PRINTCHR$(17);:NEXT
54 PRINTTAB(14)"PRESS A KEY"
55 GETJ$:IFJ$<>""THEN55
56 GETA$:IFA$=""THEN56
57 PRINTTAB(14)CHR$(145)"
60 REM MAIN LOOP
70 GETA$
80 IFA$="W"THENDY=-1:DX=0
90 IFA$="X"THENDY=1:DX=0
100 IFA$="A"THENDX=-1:DY=0
110 IFA$="D"THENDX=1:DY=0
120 IFA$="Q"THENDN=-1:DM=0
130 IFA$="M"THENDN=1:DM=0
140 IFA$="J"THENDM=-1:DN=0
150 IFA$="L"THENDM=1:DN=0
155 Y1=Y1+DY
156 IFY1<1ORY1>24THENF=0:GOSUB1000:GOTO25
157 X1=X1+DX
158 IFX1<00RX1>39THENF=0:GOSUB1000:GOTO25
160 Y2=Y2+DN
162 IFY2<1ORY2>24THENF=1:GOSUB1000:GOTO25
164 X2=X2+DM
166 IFX2<00RX2>39THENF=1:GOSUB1000:GOTO25
167 P1=X1+40*Y1
168 P2=X2+40*Y2
170 IFPEEK(SC+P1)=160THENF=0:GOSUB1000:GOTO25
180 IFPEEK(SC+P2)=160THENF=1:GOSUB1000:GOTO25
190 POKE SC+P1,160
200 POKE CO+P1,14
210 POKE SC+P2,160
220 POKE CO+P2,5
230 GOTO70:REM RESTART LOOP
240 :
1000 REM SCORE S/R
1020 IF F=1THENS1=S1+1:GOSUB2000:RETURN
1030 S2=S2+1:GOSUB2000:RETURN
1999 :
2000 REM FLASH SCREEN S/R
2010 FORJ=1TO10
2020 FORI=0TO15
2030 POKE53281,I
2040 NEXTI,J
2045 POKE53281,0
2050 RETURN
```

BBC Micro

```
10  REM BBC
20 MODE1:cycle1=0:cycle2=0
30 PROCinitialise
40 PROCkey
50 PROCborder
60 PROCscore
70 PROCplot
80 PROCkeyboard
90 PROCtest_point
100 GOTO70
110 END
120 DEF PROCborder
130 GCOL0,border
140 MOVE0,0
150 DRAW1279,0
160 DRAW1279,980
170 DRAW0,980
180 DRAW0,0
190 ENDPROC
200 DEF PROCkeyboard
210 REM PLAYER ONE.
220 IF INKEY(-51)=-1THEN dx=4:dy=0
230 IF INKEY(-87)=-1THEN dx=-4:dy=0
240 IF INKEY(-66)=-1THEN dm=-4:dn=0
250 IF INKEY(-70)=-1THEN dx=0:dy=-4
260 IF INKEY(-67)=-1THEN dx=0:dy=4
270 IF INKEY(-34)=-1THEN dm=0:dn=4
280 IF INKEY(-55)=-1THEN dm=0:dn=-4
290 IF INKEY(-102)=-1THEN dm=0:dn=-4
300 x=x+dx:y=y+dy:m=m+dm:n=n+dn
310 ENDPROC
320 DEF PROCinitialise
330 x=100:y=490:m=1179:n=490
340 dx=4:dy=0:dm=-4:dn=0
350 col1=1:col2=2:border=3
360 ENDPROC
370 DEF PROCtest_point
380 pixel1=POINT(x,y)
390 pixel2=POINT(m,n)
400 IF pixel1 THEN PROCexplode(x,y,1)
410 IF pixel2 THEN PROCexplode(m,n,2)
420 ENDPROC
430 DEF PROCplot
440 GCOL0,col1:PLOT69,x,y
450 GCOL0,col2:PLOT69,m,n
460 ENDPROC
470 DEF PROCexplode(ex,ey,which)
480 FORI=1TO100
490 MOVEex,ey
500 GCOL0,RND(3)
510 PLOT1,RND(50)-25,RND(50)-25
520 NEXT
530 IF which=2 THEN cycle1=cycle1+1
540 IF which=1 THEN cycle2=cycle2+1
550 PROCinitialise
560 CLS
570 PROCkey
580 PROCborder
590 PROCscore
600 ENDPROC
610 DEF PROCscore
620 PRINTTAB(1,0)"Cycle One="¡cycle1
630 PRINTTAB(28,0)"Cycle Two="¡cycle2
640 ENDPROC
650 DEF PROCkey
660 PRINTTAB(8,12)"PRESS ANY KEY TO START"
670 *FX21
680 A$=GET$
690 PRINTTAB(8,12)"
700 ENDPROC
```

DO THE LOGOMOTION

In this instalment of our LOGO course, we will develop a simple game in which the turtle gets lost in space. To do this, we will first need to look more closely at various input and output methods.

In our 'Space Turtle' game, the turtle is stranded in the depths of space, a long distance from its base, to which it must return. The game will require us to print various messages on the screen. The necessary command for this is, not surprisingly, PRINT. Once a message has been printed, the cursor is moved to the beginning of the next line.

To print a single word, PRINT is followed by the word itself — thus, PRINT "HELLO prints the word 'HELLO' on the screen. PRINT " is used to print the 'null word' (a 'word' that has no characters). The effect of this command is simply to print a blank line. If more than one word is to be printed, the text is enclosed in square brackets to indicate that it forms a *list*:

```
PRINT [YOUR TIME HAS RUN OUT]
```

PRINT is also used to display the contents of a variable, so PRINT :SCORE will take the value held in the variable "SCORE and display it. Messages and variable values may be combined in the same PRINT statement by enclosing the complete instruction in *round* brackets, as in:

```
(PRINT [YOUR SCORE WAS] :SCORE)
```

PRINT1 behaves in exactly the same fashion as PRINT does, except that in this case the cursor will remain at the end of the printed text and will not be moved to the next line. This can be demonstrated by entering:

```
PRINT1 [WHAT IS YOUR NAME?]
```

OUTPUT OPERATIONS

LOGO commands, such as HIDETURTLE or PRINT, cause something to happen — they may be said to have an effect on the turtle. However, other LOGO primitives — XCOR, for example — do not have an effect, but instead output a value. This value is then normally used as the input to a command. So, for example, typing:

```
PRINT XCOR
```

would cause XCOR to output the value corresponding to the turtle's current x co-ordinate to the command PRINT, which then displays the result. Thus, if the current value of XCOR is 20, PRINT XCOR will cause the number 20 to appear on the screen. If XCOR is typed on its own, the message RESULT: 20 will appear. This is actually an error message (LCSI versions are somewhat less polite and would print YOU DON'T SAY WHAT TO DO WITH 20).

The procedures we have so far written have all been commands. To create *operations* we must make use of the primitive OUTPUT. As a simple example, here's a procedure that outputs the distance of the turtle from the origin; this procedure uses SQRT to return the square root of a number:

```
TO DISTANCE
  OUTPUT SQRT (XCOR*XCOR + YCOR*YCOR)
END
```

Try moving the turtle to different screen positions and use DISTANCE to determine how far it is from the origin. For example, SETXY 30 40 PRINT DISTANCE should give the answer 50.

When LOGO executes an OUTPUT instruction it stops running the current procedure, returning control to the procedure that called it. This can be seen in the procedure MAX, which outputs the larger of two numbers:

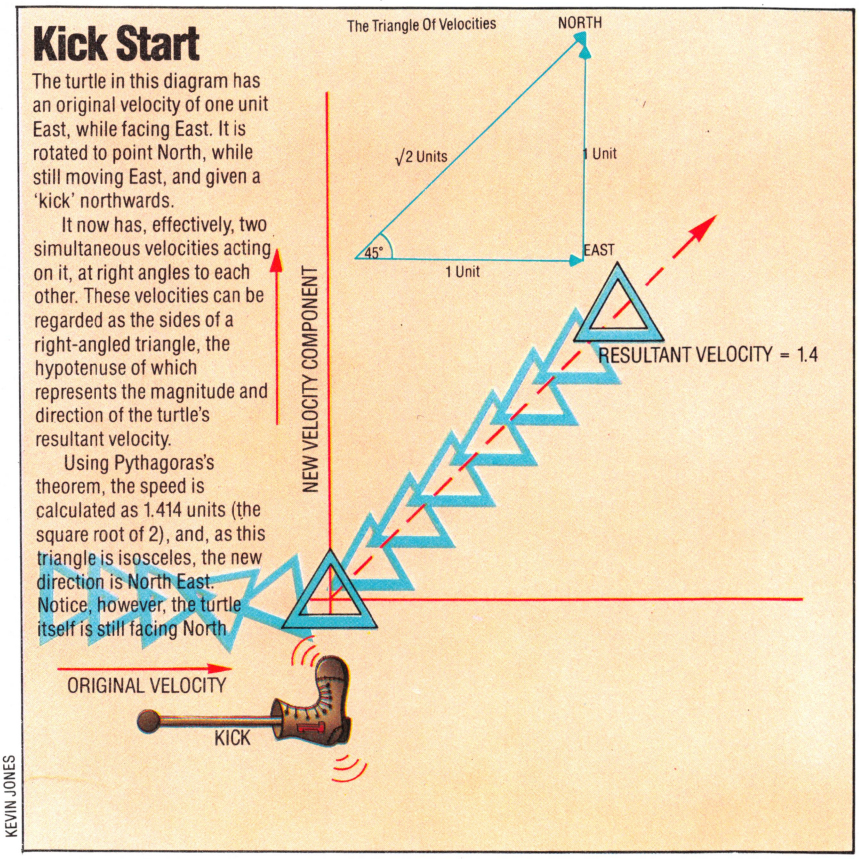
```
TO MAX :X :Y
  IF :X > :Y THEN OUTPUT :X
  OUTPUT :Y
END
```

Kick Start

The turtle in this diagram has an original velocity of one unit East, while facing East. It is rotated to point North, while still moving East, and given a 'kick' northwards.

It now has, effectively, two simultaneous velocities acting on it, at right angles to each other. These velocities can be regarded as the sides of a right-angled triangle, the hypotenuse of which represents the magnitude and direction of the turtle's resultant velocity.

Using Pythagoras's theorem, the speed is calculated as 1.414 units (the square root of 2), and, as this triangle is isosceles, the new direction is North East. Notice, however, the turtle itself is still facing North



KEVIN JONES

PRINT MAX 6 2 will give 6 as a result. Try writing a procedure to give the absolute value of a number, so that PRINT ABS 4 and PRINT ABS (-4) will both return the value 4.

Our game will ask you to type in your name and press Return. Here is a procedure to do this:

```
TO GET.NAME
   SPLITSCREEN
   PRINT1 [WHAT IS YOUR NAME?]
   MAKE "NAME FIRST REQUEST
   (PRINT "HELLO :NAME )
END
```

REQUEST waits for a line to be typed in and terminated with a Return. It then outputs the line as a list. FIRST outputs the first element of a list. Try the GET.NAME procedure and type in 'Holly' as the name. Now see what happens if 'Holly Johnson' is used as an input.

The game will control the turtle's onscreen movement by using the keys R, L and K. R will turn the turtle clockwise (right) through 30 degrees; L will turn it anticlockwise (left) by the same amount; while K is used to 'kick' the turtle — increasing its speed in whatever direction it is currently facing. The turtle will be moving around the screen, and we will require it to respond immediately to these keys. It would be a help if there was a LOGO primitive — READKEY, perhaps — that would output the last key that was pressed. If this was the case, we could write:

```
TO COMMAND
   MAKE "COM READKEY
   IF :COM = "R THEN RIGHT 30
   IF :COM = "L THEN LEFT 30
   IF :COM = "K THEN KICK
END
```

Unfortunately, this primitive does not exist! However, we can write it as a procedure, thus:

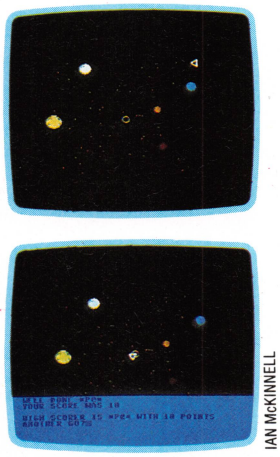
```
TO READKEY
   IF RC? THEN OUTPUT READCHARACTER
   OUTPUT "
END
```

When a key is pressed it is stored in the keyboard buffer. READCHARACTER simply outputs the last character from the buffer — if the buffer is empty READCHARACTER will wait for a key to be pressed and then output the relevant character. RC? is true if the buffer contains any characters and is false if the buffer is empty. So READKEY will now output the last character in the buffer, or will output a null word if the buffer is empty.

THE DYNATURTLE

Our space-going turtle is in fact a *dynaturtle*. This is a turtle that has a velocity, as well as a position and a heading like any normal earthbound turtle. The dynaturtle is in space, so there is no friction and no gravity. The dynaturtle will obey Newton's laws of motion. Our illustration will make this clearer, but as an example, let us assume that the dynaturtle is moving left to right across the screen with a velocity of 1. If the L key is pressed, the dynaturtle will turn to face the top of the screen, but the turtle's momentum will keep it moving on its horizontal course. If K is then pressed, the dynaturtle will get a 'kick' in the direction in which it is facing. This results in a push up the screen of velocity 1, and the dynaturtle will move diagonally across the screen with a velocity of 1.4. The dynaturtle will allow you to experiment with a body that obeys Newton's laws; it is designed to allow you to develop an intuitive understanding of these laws without you needing to understand all the relevant equations.

In the program, the dynaturtle's velocity is considered in terms of two components along the x and y axes. These components are found by using the SIN and COS functions. The only game controls are the three already mentioned. To begin the game, just type START. You have a fixed time in which to reach your goal, and the program keeps a record of the best score to date.



IAN McKINNELL

Extraterrestrial Turtle
The program as printed will generate the turtle and the target (its home base) only. The stars and planets shown here were added using some simple circle procedures

Logo Flavours

MIT LOGO	LCSI LOGO
DRAW	CS
PRINT1	TYPE
RC?	KEYP
READCHARACTER	RC
REQUEST	RL
SETHEADING	SETH
SETXY	SETPOS (followed by a list)
FULLSCREEN	FS (Not available on the Spectrum)
SPLITSCREEN	SS (Not available on the Spectrum)

IF has a slightly different syntax in LCSI LOGO, e.g:

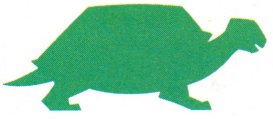
IF DISTANCE < 5 [DONE STOP]

Program Project

Write a program to play Lunar Lander. For those who do not know this game, here is a brief rundown of the objectives:
You are piloting a rocket that is some distance above the Moon's surface, and which carries a limited amount of fuel. Gravity exerts a constant pull on your craft, and increases its downward velocity by a fixed amount each second. Pressing F causes your rocket engines to fire, slowing your craft's descent but burning up more fuel. The aim of the game is to use the F key so that your descent is slow enough for a safe landing.

Abbreviations

OUTPUT	OP
PRINT	PR
READCHARACTER	RC
REQUEST	RQ
SETHEADING	SETH



Exercise Answers

1. Nested triangles

```
TO TRI :SIZE :LEVEL
   IF :LEVEL = 0 THEN REPEAT 3 [FD :SIZE RT 120]
      STOP
   TRI ( :SIZE / 2 ) ( :LEVEL — 1 )
   FD ( :SIZE / 2)
   TRI ( :SIZE / 2 ) ( :LEVEL — 1 )
   RT 60
   TRI ( :SIZE / 2 ) ( :LEVEL — 1 )
   FD ( :SIZE / 2 )
   RT 60
   TRI ( :SIZE / 2 ) ( :LEVEL — 1 )
   LT 60
   BK ( :SIZE / 2 )
   LT 60
   BK ( :SIZE / 2 )
END
```

2. Square snowflake

```
TO SNOW 1:SIZE :LEVEL
   REPEAT 4 [SIDE1 :SIZE :LEVEL RT 90]
END
TO SIDE1 :SIZE :LEVEL
   IF :LEVEL = 0 THEN FD :SIZE STOP
   SIDE1 ( :SIZE / 3 ) ( :LEVEL — 1 )
   LT 90
   SIDE1 ( :SIZE / 3 ) ( :LEVEL — 1 )
   RT 90
   SIDE1 ( :SIZE / 3 ) ( :LEVEL — 1 )
   RT 90
   SIDE1 ( :SIZE / 3 ) ( :LEVEL — 1 )
   LT 90
   SIDE1 ( :SIZE / 3 ) ( :LEVEL — 1 )
END
```

3. Curve with no gradient at any point

```
TO W :XSTEP :YSTEP :LEVEL
   WUP :XSTEP :YSTEP :LEVEL
   WDOWN :XSTEP :YSTEP :LEVEL
END

TO WUP :XSTEP :YSTEP :LEVEL
   IF :LEVEL = 0 THEN SETXY ( XCOR + :XSTEP )
      ( YCOR + :YSTEP ) STOP
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1)
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
END

TO WDOWN :XSTEP :YSTEP :LEVEL
   IF :LEVEL = 0 THEN SETXY ( XCOR + :XSTEP )
      ( YCOR — :YSTEP ) STOP
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( : LEVEL — 1)
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
   WUP ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( : LEVEL — 1 )
   WDOWN ( :XSTEP / 6 ) ( :YSTEP / 2 ) ( :LEVEL — 1 )
END
```

Space Turtle Program

```
TO START
   MAKE "MAX 0
   MAKE "BEST "
   DRAW
   HT
   TARGET
   PLAY
END

TO TARGET
   PU SETXY 0 5 PD
   RT 90
   REPEAT 36 [FD 31.4/36 RT 10]
   PU
END

TO PLAY
   GET.NAME
   INIT
   DRIVE
END

TO GET.NAME
   SPLITSCREEN
   PRINT1 [WHAT IS YOUR NAME?]
   MAKE "NAME FIRST REQUEST
END

TO INIT
   MAKE "SCORE 200
   SETXY 100 100
   SETH 270
   MAKE "XVEL 0
   MAKE "YVEL 0
   FULLSCREEN
   ST
END

TO DRIVE
   COMMAND
   DYNA.MOVE
   IF DISTANCE <5 THEN DONE STOP
   MAKE "SCORE :SCORE – 1
   IF :SCORE = 0 THEN OUT.OF.TIME
   STOP
   DRIVE
END

TO COMMAND
   MAKE "COM READKEY
   IF :COM = "R THEN RIGHT 30
   IF :COM = "L THEN LEFT 30
   IF :COM = "K THEN KICK
END

TO READKEY
   IF RC? THEN OUTPUT
   READCHARACTER
   OUTPUT "
END

TO KICK
   MAKE "XVEL + 3 * SIN HEADING
   MAKE "YVEL + 3 * COS HEADING
END

TO DYNA.MOVE
   SETXY XCOR + :XVEL
   YCOR + :YVEL
END

TO DISTANCE
   OUTPUT SORT
   (XCOR * XCOR + YCOR * YCOR)
END

TO DONE
   PRINT"
   SPLITSCREEN
   (PRINT [WELL DONE] :NAME)
   (PRINT [YOUR SCORE WAS]
   :SCORE)
   REPORT
   AGAIN
END

TO REPORT
   IF :SCORE > :MAX THEN MAKE
   "MAX :SCORE MAKE "BEST :NAME
   PRINT "
   (PRINT [HIGH SCORER IS] :BEST
   [WITH] : MAX [POINTS])
END

TO AGAIN
   PRINT1 [ANOTHER GO?]
   MAKE "ANS FIRST REQUEST
   IF :ANS = "YES THEN REPLAY
   STOP
   IF :ANS = "NO THEN STOP
   PRINT [MAKE YOUR MIND UP, YES
   OR NO?]
   AGAIN
END

TO OUT.OF.TIME
   PRINT "
   SPLITSCREEN
   PRINT [YOUR TIME HAS RUN OUT]
   AGAIN
END

TO REPLAY
   HT
   GET.NAME
   INIT
   DRIVE
END
```

RISING TO ZERO

We have now had a fairly thorough look at the addressing modes available on the 6809 processor, particularly the use of indirect addressing. There are still some variations we will need to discuss in more detail in the course, most notably the use of the program counter in indexing. For the moment, let's take a closer look at how the stacks are used.

So far in the course, we have used the two stack pointer registers, S and U, only as extra index registers. The use of the so-called 'hardware stack' for the storage of return addresses on subroutine calls has also been mentioned, although only in passing. Now we need to backtrack a little and consider the architecture of a stack, and the way it is used.

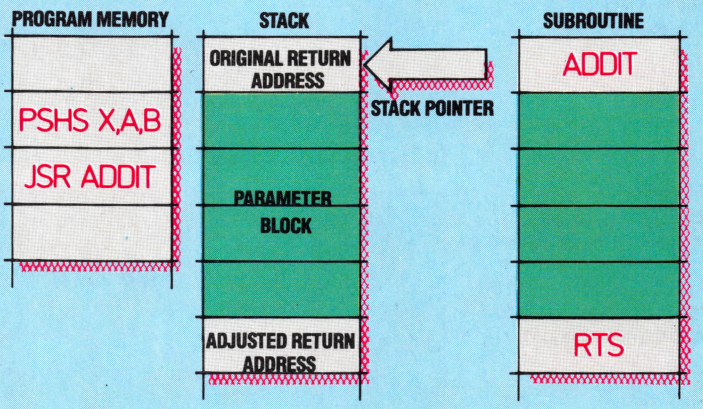
A stack is a special instance of a more general type of data arrangement known as a *list*. You should be familiar with the everyday idea of a list, even if you know little about the increasingly popular list processing languages, such as LISP and LOGO. A list is simply a sequence of data items. This sequence can be arranged in an order determined by some property of the data (for example, a series of numbers in numerical order, or a string of characters in alphabetical order), or it can be a random arrangement determined by the order in which data items were added to the list. With all of these lists it is sensible to attach significance to the identity or value of the 'next' or the 'previous' item in the list, and particularly to the list's first item (known as its 'head') and its last item (the 'tail').

One important feature of a list is that it is a *dynamic data structure*; that is to say, items of data can be added to, or taken from, the list at will. In a general list, data can be added or removed at any position in the list. The particular restriction that specifies that a given list is a *stack* is that data can be added to, or taken from, a stack only at one end. Each new item added to a stack becomes the 'listhead', and only this can be removed from it.

The name itself gives a good idea of the way a stack operates. Consider a stack of plates in a canteen: as a plate is needed it is taken from the top, and clean plates are put only on the top of the stack. You could add plates to, or take them from, the middle of the stack, but this would be unnecessarily problematic. It is possible, however, to inspect an item anywhere in the stack.

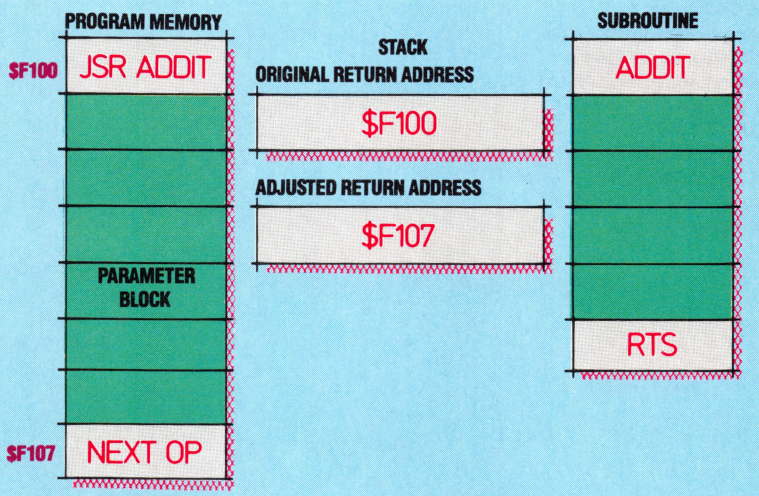
There are two extreme situations that can arise when a stack is operating: either the stack becomes empty, which is no problem if the next stack operation adds an item to it, but could be awkward otherwise; or alternatively, the stack could fill to overflowing. This second situation can be better visualised if we consider our stack of plates in a canteen: there would come a point where the stack

Parameter Push



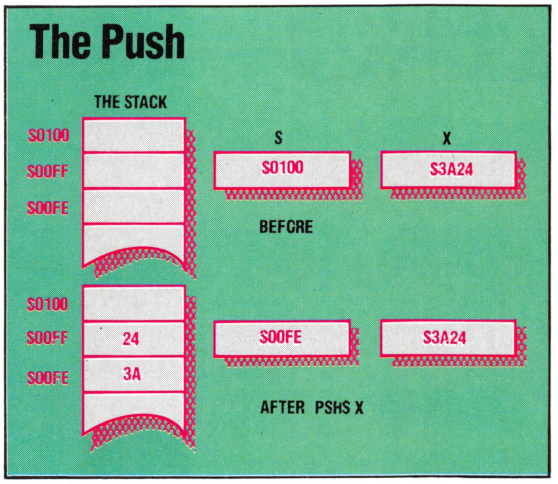
Parameters can be passed to a subroutine by loading them into registers and then pushing them onto the stack. The subroutine can pull them off the stack, taking care to move the JSR return address down the stack when the parameters have been accessed. If this is not done, then the stack will grow continually, and eventually overflow

Parameter Insertion

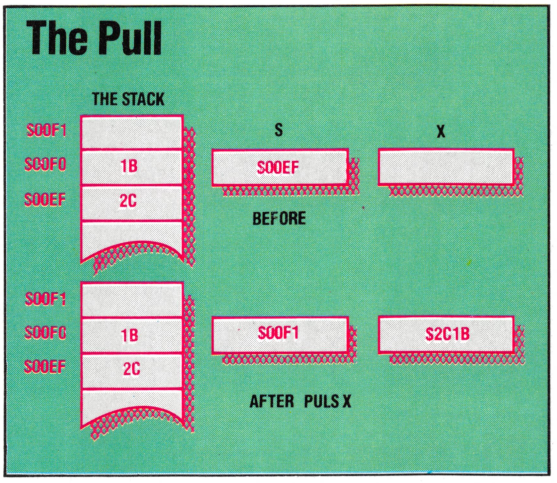


A more usable method of passing parameters is to insert them into the program directly after the JSR call to the handling subroutine. The subroutine can then use the return address on the stack as the base address of the parameter block, and access it by indexed addressing. The return address must then be adjusted to point to the next program instruction, rather than to the start of the parameter block

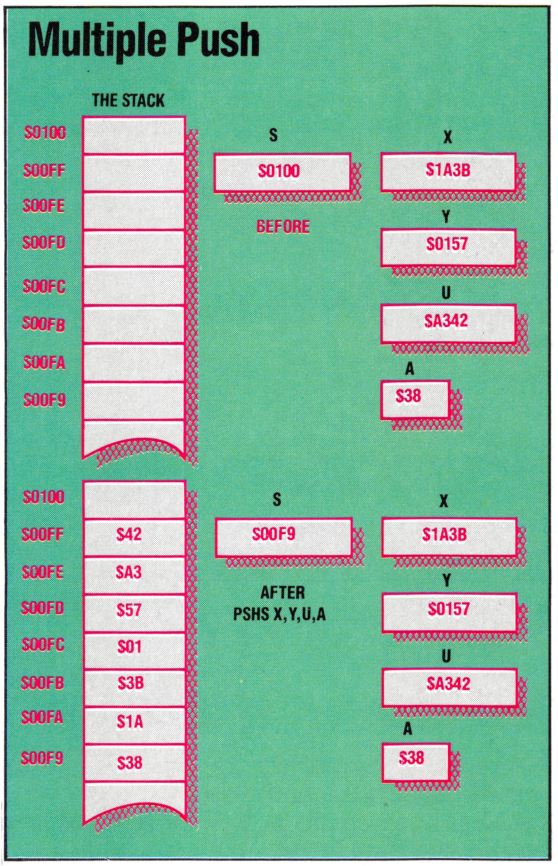
Push Off
The 6809 stack pointer always addresses the 'top' of the stack — that is, the byte most recently written to. When a PSHS X is executed, therefore, S is decremented by two, so that it points to the new stacktop, and the contents of X (a two-byte register) are then written at that address in hi-lo format. Notice that the stack 'rises to zero' — the stack pointer points to lower locations in memory as the stack grows



Pull Together
When PULS X is executed, the contents of the two bytes at the current stack pointer address are copied to X, and S is then incremented by two to point to the new stacktop



Push Off Together
When a multiple-register stack operation is executed, the registers involved are accessed in a pre-determined order — PC,U or S,Y,X,DP,B,A,CC. When PSHS X,Y,U,A is executed, therefore, the contents of U are stacked first, followed by Y, X and A



LIZ DIXON

of plates reached the ceiling, and no more could be added to it.

Stacks in computers work in much the same way. The two operations of adding and removing items are known as *pushing* and *pulling* (or *popping*), respectively. The two extreme situations we have just mentioned are referred to as *underflow* and *overflow.*

Stacks can be implemented in a number of ways (using arrays in a BASIC program, for example), but the method that we are considering requires a block of available memory and a register that we can designate the *stack pointer.* This pointer is necessary to keep track of the current location of the listhead. Unlike a stack of plates, a memory stack cannot be assessed by inspection since there is nothing to distinguish a memory location containing an item of stack data from the next location, which may not be part of the stack. It's worthwhile pointing out that, just as data is not really 'loaded' from memory into a register but only copied, so similarly items are not really 'pulled' off a stack — only the pointer to the top of the stack is changed.

The stack pointer, therefore, contains the address of the current top of the stack. There are two variations possible here: the stack pointer can give either the address of the next free location where data can be stored, or it can give the address of the last item of data stored in the stack. This latter is the convention used by the 6809 processor, although there is no particular advantage in this over the former method — other processors use that technique just as readily.

A significant difference of organisation between a memory stack and a stack of plates in a canteen is that the former grows downwards in the 6809 system: as more items are pushed onto the stack, the stack pointer address gets lower and lower—it is said to 'rise towards zero'.

STACK OPERATIONS

The two 6809 stack operations are represented by the instructions PSH, to push data onto the stack, and PUL, to pull it off. These operations can be applied to either of the two pointers, S and U, so we have PSHS, PULS, PSHU and PULU. The data that is operated on must come from, or go to, a register, although a number of registers can be pushed or pulled in one instruction.

The instruction PSHS X will have the effect of *first* decrementing S, the stack pointer, by two (or one if an eight-bit register is pushed) to give the address of the next free stack location, and *second* storing the contents of X at that address. The first diagram illustrates this procedure. Notice again the 6809 hi-lo addressing convention: the hi-byte ($3A) of X is stored at $00FE , a lower position in memory than the lo-byte ($24), which is stored at $00FF. If you use an assembler, these details of whether stack pointers increment or decrement are irrelevant — the assembler does all the memory management necessary.

The instruction PULS X has the opposite effect:

the 16-bit value at the address currently in S is loaded into X, and the contents of S are then incremented by two. The second diagram shows these changes.

More than one register can be pushed or pulled at a time. Consider the instruction:

```
PSHS X,Y,U,A
```

When more than one register is pushed like this, the order in which the registers are listed is ignored, and instead the registers are always pushed in this order: PC (the program counter register), U or S, Y, X, DP (the direct page register), B, A and CC (the condition code register). They will, of course, be pulled off in the reverse order. The only real constraint on stack operations is that neither S nor U can be pushed onto its own stack.

The stacks are used in general programming as convenient places for fast, temporary storage, but their major uses come when dealing with interrupts (more about these later in the course) and subroutines. We have already seen how the contents of the program counter register are automatically pushed onto the stack when a subroutine is called, and pulled on return from the subroutine (RTS is equivalent to PULS PC). Either stack, but particularly S, can also be used to pass parameters to a subroutine.

The method we have used so far for passing parameters via the registers (as in the Jump Table program on page 639) has two major weaknesses. First of all, there may be more parameters to pass than there are registers, and, secondly, it can prove awkward when the routine called uses a register holding a parameter that you need to retain. There are, however, two other common techniques for passing parameters:

1) The data can be stored in the middle of the program by using FCB, FDB or FCC directives immediately after the subroutine call. The value of the program counter register pushed onto the stack by the JSR instruction gives the address of the first of these values (since PC always points to the next byte after the current instruction), and can be used to obtain all of them, with suitable offsets. The first example program illustrates this technique. Care must be taken to arrange the RTS instruction so that it passes control to a real instruction, and not to an item of data.

2) The data can be loaded into registers and pushed onto the stack before the subroutine call, from which it can be pulled into the subroutine and used. Care must be taken here that, at the RTS instruction, the stack pointer will access the previously stacked PC return address. The second piece of code illustrates this technique. This is generally a more useful method than the first.

In both methods, the dual role of S and U as index registers as well as stack pointers means that items on the stack can be referenced by indexed addressing in addition to being easily accessed for removal from the stack. This makes it easier to ensure that the correct items are left on the stack for the return.

Multiple-Precision Addition

Here are two pieces of code showing alternative methods of performing multiple-precision addition using the stacks. In the first piece of code, the parameters are placed after the subroutine call. A typical call to add two four-byte numbers at $100 and $104, leaving the result at $108 would be:

```
        BSR     MPADD
        FCB     4           Length of each number (in bytes)
        FDB     $100        Address of first number
        FDB     $104        Address of second number
        FDB     $108        Address of result
        ORG     $1000
MPADD   PSHS    X,Y,U,A,B,CC  Save all registers. This pushes nine bytes onto the stack
        LDU     9,S         This instruction will load into U the PC return address pushed onto the stack by the BSR instruction
        PULU    X,Y,B       Treating the data that comes after the subroutine call as though it were a stack, we pull the addresses of the two numbers into X and Y, and the length into B
        LDU     ,U          The address of the result goes into U
        ANDCC   #%11111110  Clear the carry flag
LOOP    LDA     ,X+         Perform the addition of one byte to the other, adding with carry
        ADCA    ,Y+
        STA     ,U+         Store the result
        DECB                Check if the addition is finished
        BNE     LOOP        If not, LOOP
        LDU     9,S         Else get return address and add seven to skip over the seven bytes of parameters stored after the call
        LEAU    7,U
        STU     9,S
        PULS    PC,U,X,Y,A,B,CC  Restore all the registers to the state that they were in before, restoring the PC instead of using an extra RTS
```

The second example performs the same operation but pushes parameters onto the stack. The calling sequence would be:

```
        LDU     #$108       Result address
        LDX     #$100       Operand address
        LDY     #$104
        LDA     #4          Length
        PSHS    U,X,Y,A     Put parameters onto stack
        BSR     MPADD
        ORG     $1000
MPADD   PSHS    X,Y,U,A,B,CC  Save all registers
        LEAU    11,U        Saving the registers takes nine bytes, and the return address takes two bytes — a total of eleven in all. U now points to the parameters
        PULU    X,Y,B
        ANDCC   #%11111110
LOOP    LDA     ,X+         Proceed as before
        ADCA    ,Y+
        STA     ,U+
        DECB
        BNE     LOOP
        PULS    PC,U,X,Y,A,B,CC
```

FOUR-MINUTE WARNING

Perhaps one of the most bloodthirsty (some would say tasteless) games yet designed, Missile Command is guaranteed to send shivers down the spines of anti-nuclear campaigners. The game consists of defending civilisation against waves of multiple-warhead nuclear missiles.

The success of a game designed for an amusement arcade oftens depends on the persuasiveness of its 'attract' mode (the display that appears on the screen while the game is not being played). The game must also be instantly addictive. A game that fails on either count will soon be removed by the arcade owner in favour of a more profitable machine. Although Missile Command is now past its peak in the arcade, for a time it was highly successful. The version available for the Atari machines bears witness to its former glory.

The scenario of the game is both simple and subtle. The player is put in the position of the commander of an anti-missile station during a nuclear war and must protect six cities from destruction by exploding nuclear anti-missiles in the path of the incoming warheads. This is designed to appeal to both the megalomania and gallantry latent in a 'shoot-em-up' games addict — whether in an arcade or at home.

In play, the screen shows a cross-section view of the action, depicting the six cities and a pyramid structure in the centre with the anti-missiles ready for launch. The tracks of the incoming missiles then appear from the top of the screen. The player moves a cross around the screen with the joystick. The cross is positioned in the path of the incoming missiles and, by pressing the fire button, an anti-missile is launched from the player's missile base. This explodes at the co-ordinates of the cross, destroying all of the incoming missiles within range of the explosion.

However, a number of the enemy missiles have multiple warheads that split into several tracks, each of which can destroy a whole city. The game is further complicated by the appearance of low-flying enemy aircraft and satellites, all capable of releasing waves of missiles. Points are awarded for the destruction of enemy aircraft and missiles. At the end of an attack wave, a pictogram shows the number of cities you have successfully defended from annihilation and the number of anti-missiles that you have left.

As the player progresses through the various levels of the game, the attacking missiles begin to move faster and the number of warheads they split into increases. At this point it is necessary to develop an overall strategy, rather than simply picking off each missile separately. The player may choose, for example, to lay down a 'barrage' of anti-missiles, which will explode in a line and with luck destroy the wave in one swoop.

The game is further complicated at the higher levels by the appearance of parachute-borne warheads. These are extremely difficult to destroy: if your missile explodes slightly off-target, it is likely to be 'blown' out of the way (presumably on the updraught), and therefore a successful missile needs to be exploded directly on top of one of these.

Throughout the game, you must remember that you have only a limited number of anti-missiles (30 for the first level) and if these are squandered you have to look on helplessly as the enemy rockets annihilate the missile base and the cities.

Each level consists of two separate attack waves, after which the score is computed. Like other Atari games, it is possible to 'skip' to a higher level of the game. Each level is distinguished by different foreground and background colours. The game ends when all six cities are destroyed, and this intrinsically pessimistic conclusion is reinforced by a final screen displaying a suitably apocalyptic explosion and the words 'THE END'.

For the Atari computers, the game is available on cartridge, and comes with a large colour brochure that easily outclasses the documentation supplied with most other games software. The booklet gives detailed descriptions on how to set up the game, point-scoring and hints on methods of play, as well as colour illustrations.

Missile Command: For all Atari computers, £9.99
Publishers: Atari Corporation UK, Ltd, Atari House, Railway Terrace, Slough, Berkshire
Authors: Atari
Joysticks: Required
Format: Cartridge

LIZ DIXON

DATABASE

Here, courtesy of Zilog Inc., we produce another part of the Z80 programmers' reference card.

8-Bit Arithmetic and Logical Group

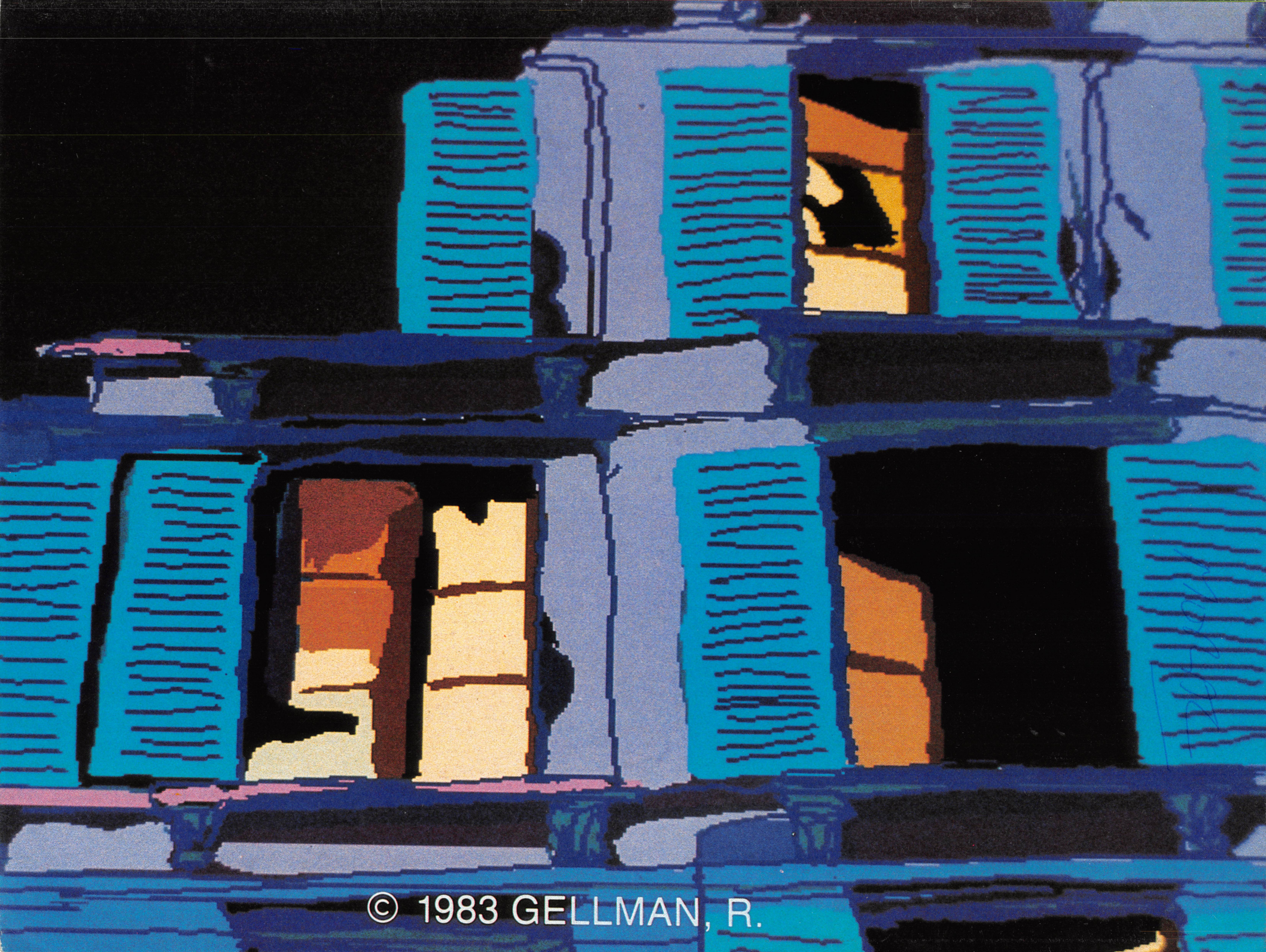
SOURCE

	REGISTER ADDRESSING							REG INDIR	INDEXED		IMMED
	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)	n
ADD	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY ADC	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT SUB	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY SBC	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
AND	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
XOR	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
OR	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE CP	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT INC	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT DEC	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
ADD A, r	A ← A + r	↕	↕	X	↕	V	0	↕	10 [000] r		1	1	4	r Reg
ADD A, n	A ← A + n	↕	↕	X	↕	V	0	↕	11 [000] 110 ← n →		2	2	7	000 B 001 C 010 D
ADD A, (HL)	A ← A + (HL)	↕	↕	X	↕	V	0	↕	10 [000] 110		1	2	7	011 E
ADD A, (IX + d)	A ← A + (IX + d)	↕	↕	X	↕	V	0	↕	11 011 101 10 [000] 110 ← d →	DD	3	5	19	100 H 101 L 111 A
ADD A, (IY + d)	A ← A + (IY + d)	↕	↕	X	↕	V	0	↕	11 111 101 10 [000] 110 ← d →	FD	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	X	↕	V	0	↕	[001]					s is any of r, n, (HL), (IX + d), (IY + d) as shown for ADD instruction. The indicated bits replace the [000] in the ADD set above.
SUB s	A ← A − s	↕	↕	X	↕	V	1	↕	[010]					
SBC A, s	A ← A − s − CY	↕	↕	X	↕	V	1	↕	[011]					
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	[100]				
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	[110]				
XOR s	A ← A ⊕ s	↕	↕	X	0	X	P	0	0	[101]				
CP s	A − s	↕	↕	X	↕	X	V	1	↕	[111]				
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r [100]		1	1	4	
INC (HL)	(HL) ← (HL) + 1	↕	↕	X	↕	X	V	0	•	00 110 [100]		1	3	11	
INC (IX + d)	(IX + d) ← (IX + d) + 1	↕	↕	X	↕	X	V	0	•	11 011 101 00 110 [100] ← d →	DD	3	6	23	
INC (IY + d)	(IY + d) ← (IY + d) + 1	↕	↕	X	↕	X	V	0	•	11 111 101 00 110 [100] ← d →	FD	3	6	23	
DEC m	m ← m − 1	↕	↕	X	↕	X	V	1	•	[101]					m is any of r, (HL), (IX + d), (IY + d) as shown for INC. DEC same format and states as INC. Replace [100] with [101] in opcode.

NOTES: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.


© 1983 GELLMAN, R.